universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Central Checkout System for the SMILE mission“

verfasst von / submitted by

Dominik Möslinger, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2021 / Vienna, 2021

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066861
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Astronomie UG20
Betreut von / Supervisor:	Univ.-Prof. Mag. Dr. Franz Kerschbaum

Acknowledgement

At first, I want to thank my parents, Franz and Heidi Möslinger, for their endless support, encouragement and faith in me. They gave me the opportunity to attend university and have always been their for me, if I needed any sort of advice. My sister Anna Möslinger for taking time and proofreading this work.

Special thanks go to my partner, Julja Haller, for the love, motivation, aid and kindness I receive from you every day.

I also want to thank the whole group of Franz Kerschbaum and Roland Ottensamer, for the assistance I got during the development of this work. Especially, Marko Mecina, who developed the CCS-CHEOPS and spent a lot of time introducing me to the program, as well as helping me with many problems that I encountered during the development of the SMILE-CCS.

Contents

1 Introduction to SMILE

1.1 General Information

The Solar wind Magnetosphere Ionosphere Link Explorer (SMILE, Fig. 1.1) is a joint mission between the CAS (Chinese Academy of Sciences) and the ESA (European Space Agency) with the purpose of increasing our knowledge of the interaction between the Earth's dayside magnetic field and the solar wind. It is currently scheduled to launch in November 2024 as either the primary passenger on a Vega-C or as a co-passenger on an Ariane 6 with a nominal duration of 3 years. However, the mission could easily be lengthened since all needed power is generated by the satellite itself. Additionally, the cooling process is passive, therefore no additional cooling medium is needed, which could run out if the mission is prolonged. The mass for the start is 2200 kg, which consists of the propellant (1520 kg), the platform (547 kg) and the payload module (132 kg). SMILE will be an ESA-S class mission and has therefore a financial limit of 50 million Euros.

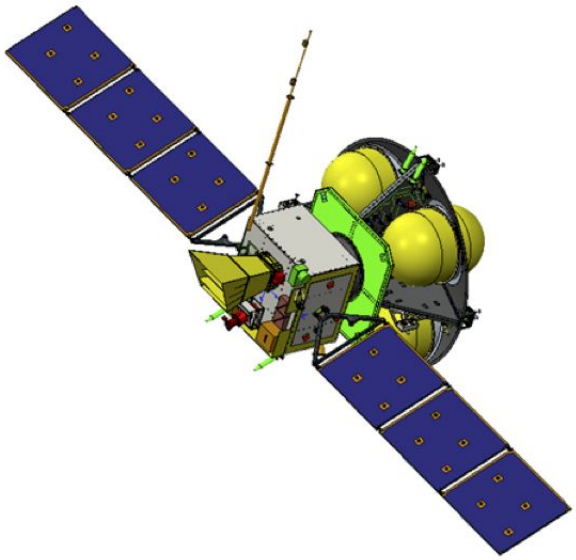

Figure 1: An artwork of the SMILE design (*Airbus Website*).

It will be brought to a highly elliptic and inclined (<70°) Orbit, with a maximum distance to Earth of 121 182 km which is about a third of the way to the Moon or 19 Earth radii. One orbit will take about 51 hours, with up to 40 hours of continuous scientific observations (*ESA Website/ SMILE mission*). This generates a lot of data which has to be somehow transmitted. As seen in Figure 2, a small part of each orbit is reserved for that. The

connection will be in the X-Band, which has a very high data rate of 36 Gbit/orbit. It will be also possible to communicate with the satellite via the much slower S-Band (22Mb/day), which is used to get the general Housekeeping (HK) data, that reports the general state of the satellite, and to send Telecommands (TC).

The data will be produced by four on-board instruments (detailed information in Section 1.2):

Soft X-Ray Imager (SXI): spectrally map the location, shape, and motion of Earth's magnetospheric boundaries

Ultraviolet Imager (UVI): observe global auroral distribution

Magnetometer (MAG): used to determine the orientation and magnitude of the magnetic field

Light Ion Analyser (LIA): determine the properties and behaviour of the solar wind and magnetosheath ions under various conditions

The SXI and UVI can point at these targets continuously, since the spacecraft will be three-axis stabilized.

The mission is centered around the three open scientific questions (see Section 1.3 for further information):

- What are the fundamental modes of the dayside solar wind/ magnetosphere interaction?
- What defines the substorm cycle?
- How do coronal mass ejection-driven storms arise and what is their relationship to substorms?

One of the fundamental aspects of heliophysics is the interaction between Earth's magnetosphere and the solar wind and the effects that result out of this. Earlier missions, such as THEMIS, Swarm, Cluster and Magnetospheric Multi-Scale observed the fundamental microscale processes of this systems, like energy and mass transport and the interaction between plasma and neutral regions. The only thing still missing is a global observation of the interaction.

In order to get such a global observation, all four SMILE instruments have to work simultaneously to detect the effect of solar wind-magnetosphere coupling. To observe it, observations of the global auroral distribution (UV images), X-ray images of the magnetic cusps and magnetosheath and in situ

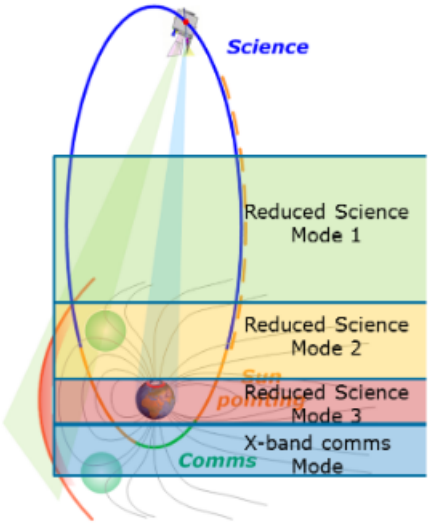

Figure 2: Different modes during one orbit. Background: pointing of SXI (green beam) and UVI (blue beam) (Branduardi-Raymont, 2018).

solar wind/magnetosheath plasma and magnetic field measurements are necessary (Branduardi-Raymont, 2018). With this, and complementary information of earlier space and ground-based observations, it will be possible to derive the first full chain of events that drive the Sun-Earth connection and gain a complete understanding of how Earth's plasma environment is controlled and formed by the Sun.

1.2 Instruments

As mentioned in Section 1.1, SMILE will simultaneously observe with four different instruments, but it is not possible for the full orbit. Figure 2 shows the five different modes which are used during a regular orbit (Branduardi-Raymont, 2018):

- *Full Science mode:* All four instruments are working.
- *Reduced Science mode 1:* UVI is not operating due to Sun exclusion constraints.
- *Reduced Science mode 2:* SXI not operating, due to Sun exclusion constraints or if the altitude is lower than 50.000 km. This is a safety mechanism to limit the radiation hitting the CCD detector of SXI.

- *Reduced Science mode 3:* Only LIA and MAG are operating.
- *X-band comms mode:* All collected scientific and telemetry data is transmitted in the X-band by the PLM to the ground.

Although the combination of in-situ and imaging instruments seams quite innovative, only state-of-the-art technology is used for this mission. Therefore very little development has to be made.

1.2.1 Soft X-Ray Imager (SXI)

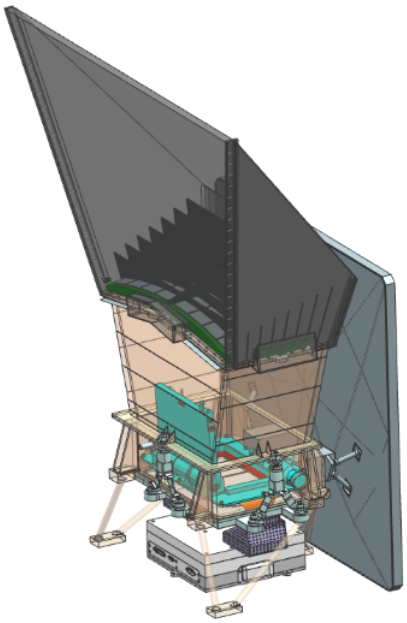

Figure 3: CAD model of the SXI instrument (Branduardi-Raymont, 2018)

SXI (model in Figure 3) is developed mainly by the University of Leicester with strong contributions from Austria (University of Vienna: instrument flight software; IWF Graz: data processing unit hardware). SXI will observe the location, shape and motion of dayside magnetospheric boundaries, including the bow shock, magnetopause and cusp regions in the soft X-ray band. It will also measure the time-dependent solar wind composition from the brightness of the solar wind charge exchange (SWCX) X-ray emission lines (Branduardi-Raymont, 2018).

SWCX describes a process, which transfers an electron from a neutral atom or molecule to high charge state heavy solar wind ion. The transferred electron is now in an excited state. As soon as the electron transits to a lower state a photon is emitted in the extreme ultraviolet (EUV) to soft X-ray range. This phenomenon has been observed at many different objects throughout our solar system, as for the Earth the neutral atom is hydrogen (Robertson et al., 2012) . This process was first modeled by Robertson and Cravens (2003) and has since been described in more detail in many papers (for example: Bhardwaj et al., 2007 and Krasnopolsky et al., 2004).

Telescope design: The entrance for the light is through the telescopes straylight baffle system (black tube in Figure 3), which includes internal

baffle vanes to minimize the straylight of the Earth and the Sun (Branduardi-Raymont, 2018).

Afterwards the light beam goes through the Radiation Shutter Assembly (turquoise in Figure 3), that closes if the radiation would get to high for the CCD. It closes during every orbit of the spacecraft for about nine hours, while the altitude is below 50.000 km. It may also be closed if the telescope changes its pointing direction to avoid direct observations of the Sun. And lastly in the event of an extremely strong solar particle event detection it may be closed to avoid further damage to the CCD.

Subsequently there is the micro pore optics (MPO) in a Lobster-eye configuration. A MPO acts like a lens in classic optical telescope. It consists of lots of thin reflecting plates. These plates form tubes in the shape of a lens to focus the light onto the CCD. The SXI system, however, is not a basic MPO system, but a Lobster-eye configuration. The idea is still similar, but the design is slightly different. A schematic illustration of the design can be seen in the top panel of Figure 4. The plates now don't form a lens, but a curved plate. This also focuses the light onto the CCD. For SXI, the whole plates will be 4 x 4 cm and have a curvature of 600 mm, which results in a 300 mm focal length. Each plate is 1.2 mm thick and the channels have a width of 40 μm, which gives a 60% fraction of open area. The channel walls are coated with iridium to maximise the reflection (Wallace et al., 2005).

At the end of the optical path are two e2v CCD370 devices. These are similar to the CCDs used for ESAs L-class mission PLATO, but optimized for SMILE's spectral range. They operate with a pixel size of 18 μm, but will be binned with a factor of 6. This leads to an effective pixel size of 108 μm (Branduardi-Raymont, 2018). They will be thermally disconnected from the telescope tube and be cooled by the Thermal Control System (TCS) to a temperature, during scientific observations, between -95 and -120°C. The TCS is a passive system which connects the CCDs via two thermal strips with the radiator.

The whole instrument has a nominal mass of 35.99 kg and is using between 17-71W, where the normal operational power should be at 51W. The field of view is with 26.5 x 15.5° quite large.

During the testing of SXI it is important to not only take into account the strength of the SWCX, but the strength of the astrophysical soft X-ray background and the particle induced background. In Table 1, the time to accomplish a SNR = 3 has been calculated for three different SW Flux strengths. In the band between 0.15-1.1 keV, where the SWCX is strongest, the main background source is astrophysical X-rays. And as it is to be expected, the time to get usable observations gets lower as the SW Flux increases. Lastly, SXI will be able to meet the scientific requirements, which

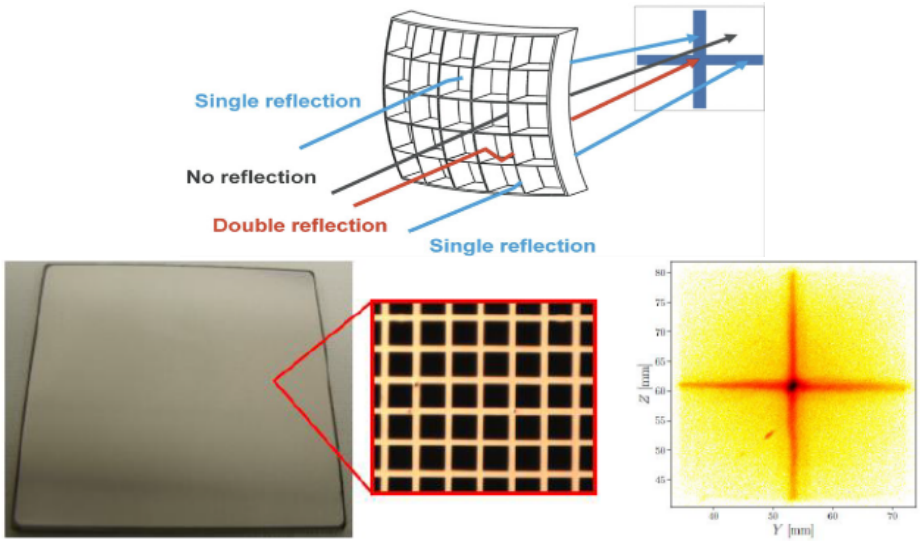

Figure 4: Top: Lobster-eye MPO system design as it is in SXI (*CAS Website*). Bottom-Left: picture of the MPO system for SXI (Branduardi-Raymont, 2018). Bottom-Right: PSF for the SXI MPO (Branduardi-Raymont, 2018).

has been shown in various tests (Branduardi-Raymont, 2018).

1.2.2 Ultraviolet Imager (UVI)

The UVI will do simultaneous observations with SXI and will monitor the Earth's northern aurora. This will link the processes of solar wind particles entering the magnetosphere (Section 1.2.1, observations by SXI) with the ones affecting the particles on the way to the aurora. To observe the aurora even in the sunlight, newly developed filters and four mirrors are used. In total UVI will have an nominal mass of 15,73 kg and is operating in a power level between 15.9 and 24.9 W. The FoV is 10 x 10° and has a spatial resolution of 150 km, when in a distance of $15 R_E$.

Optical Design: The design is based on an on-axis four mirror design as can be seen in Figure 6. The four reflecting surfaces coupled with the recent improvement of UV Filters lead to a much higher visible light suppression than earlier auroral missions could do. Four-mirror on-axis telescopes have been shown to outperform off-axis telescopes in light gathering per unit mass. With the mirror system optimized for the SMILE orbit, UVI will be able to meet all scientific requirements. At the entrance a baffle is mounted to minimize the entrance of straylight into the optical aperture. Furthermore, there is an "open-once" cover mounted, which will protect the instrument

SW Flux ($cm^{-2}s^{-1}$)	**2.0 x 10^8**	**4.9 x 10^8**	**1.5 x 10^9**
SWCX rate	0.024	0.097	0.41
SXRB rate	0.078	0.075	0.075
PB rate	0.0012	0.0012	0.0012
Time (s)	1572	167	26

Table 1: Time to accomplish a SNR = 3 for a 1° in the SXI FOV for three different solar wind strengths, SXRB (Soft X-ray Background Signal), PB (Particle Background), rate is given in counts per pixel per second integrated over the energy band 0.15 to 1.1 keV (Branduardi-Raymont, 2018)

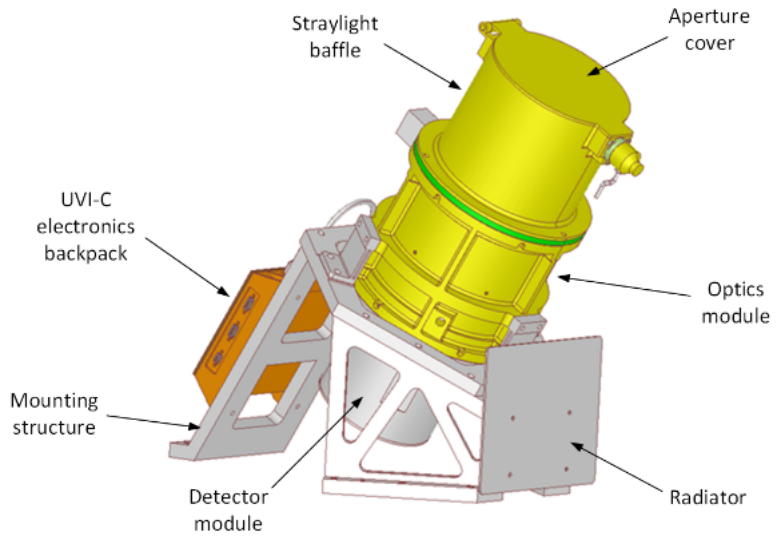

Figure 5: Model of the UVI instrument (Branduardi-Raymont, 2018)

during take off and orbital changes until the final orbit is reached. After being opened, it will stay open for the remainder of the mission.

Detector Design: Between the optical and the sealed detection system (model in Figure 7) is a 5 mm thick magnesium fluoride window, which has been tested before and is working well for auroral UV observations. The whole system is cooled by the passive UVI-Camera thermal control to a temperature range of -5°C ± 5°C. Therefore a passive radiator is attached to the instrument (Figure 5). Inside the system the first component is a photo cathode, which is photosensitive and absorbs the photon energy, which then causes an electron emission. The electrons then travel through the micro channel plate (MCP) where they get multiplied. A MCP is working with

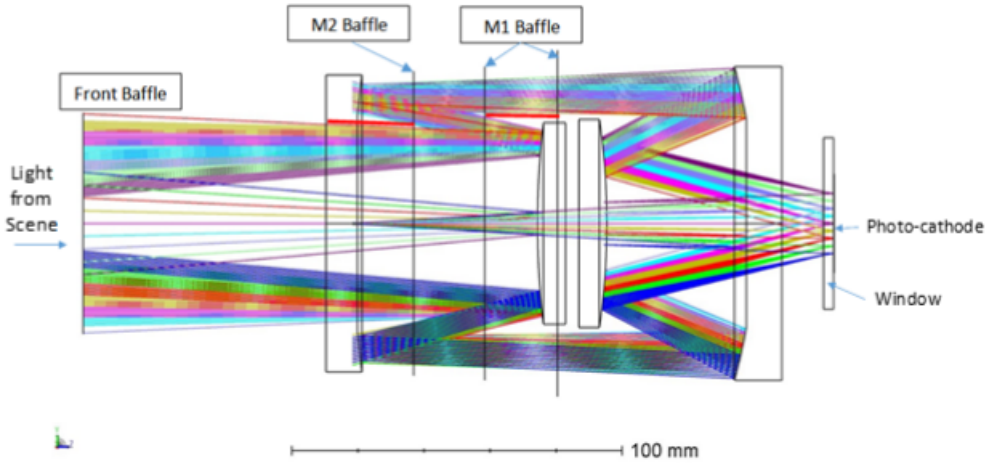

Figure 6: Model of the UVI optical system, where the light enters from the left (Branduardi-Raymont, 2018).

the same principal as a normal electron multiplier, but has a lot of small multipliers next to each other, which adds a spatial resolution. Afterwards, the electrons are converted back to light again with an aluminium coated phosphor layer. Via a fibre taper the photons reach the 1024 x 1024 pixels STAR1000 CMOS sensor, where they are finally detected.

Furthermore, it is very important to use a specific spectral band, if observations of the aurora on the dayside are done. It has to be possible to distinguish between the aurora from Earth's albedo and all nominal dayglow processes. There are two bands which could be used: The Lyman-Birge-Hopfield Long (LBH-L) band and Lyman-Birge-Hopfield Short (LBH-S) band. The LBH-L band (160-180 nm) was chosen for different reasons. First, it has often been used in prior missions, which makes it easier to base instrument models on. Second, the LBH-S band can get easier overwhelmed with the dayglow signal when the incident flux level is low.

1.2.3 Magnetometer (MAG)

In difference to the two previously described instruments, MAG (model in Figure 8) will be in-situ and is developed by the CAS with contributions from Austria. It consists of two digital fluxgate magnetometers with three ring-core sensors. Since they are orientated along three orthogonal axes, all three spatial components of the field can be measured. Both are mounted on an extendable boom, which is folded twice during launch, but will be

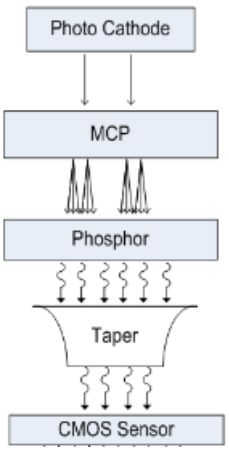

Figure 7: Model of the UVI detector system (Branduardi-Raymont, 2018).

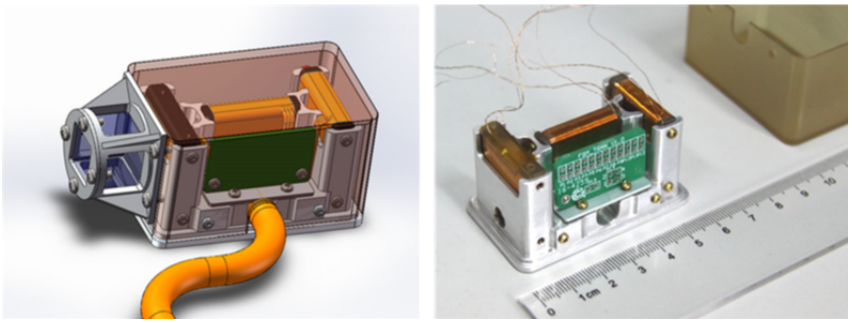

Figure 8: Model of the MAG detector which will be mounted on an extendable boom (Branduardi-Raymont, 2018).

deployed once the satellite reaches the final orbit. One instrument sits at the tip of the boom 3028 mm away from the satellite, whereas the second is mounted at 2185 mm distance. The total mass of the instrument is 8.7 kg, whereas the boom is the major contributor. All major MAG measurement parameters are listed in Table 2.

A number of error sources have to be considered when the accuracy of the instrument is calculated: sensor offset, sensor orientation, system thermal drift and errors caused by the DC (stable) and AC (changing) magnetic fields originating from the spacecraft itself. Most errors can be reduced and minimized by doing suitable ground calibrations such as sensor offset and system thermal drift. Other errors can be reduced by in flight calibrations like: sensor orientation and the DC magnetic field origination from the spacecraft. But, the AC magnetic field error can only be reduced (80%) by using both sensors. The scientifically required accuracy of 2 nT could not be achieved

Dynamic range	±12800 nT
Digital resolution	24 bit
Sampling rate	40Hz
Accuracy	1.64 nT

Table 2: Major MAG parameters

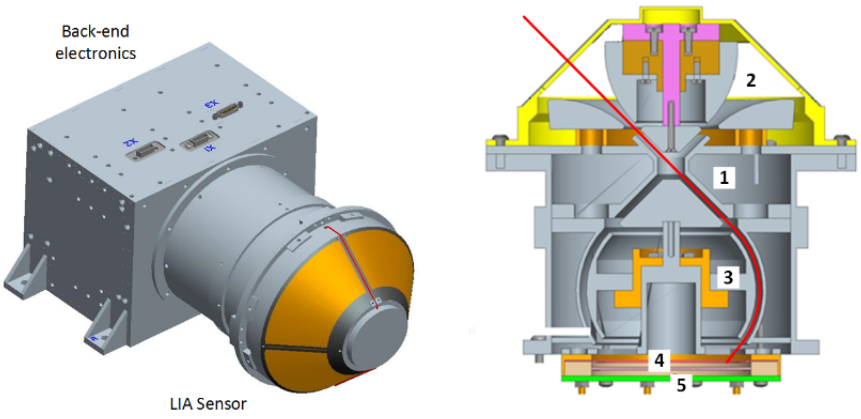

Figure 9: Left: computer model of the LIA instrument Right: optical design of LIA (Branduardi-Raymont, 2018).

with one sensor and simple calibrations at the ground, which would lead to an error of 8.58 nT. Using the in flight calibration reduces the error to 3.74 nT and only by using the second sensor the accuracy is below 2 nT at 1.64 nT. Therefore, it is important that both sensors work simultaneously.

1.2.4 Light Ion Analyser (LIA)

LIA (model in Figure 9) is mainly developed by the CAS with contributions from the UK and France. As MAG, LIA is an in-situ instrument with the goal to measure the ion distribution in the solar wind and in the magnetosheath, as well as the basic properties like density, velocity, temperature tensor and the heat flux vector. There will be two identical instruments mounted on opposite sides of the spacecraft platform, which allows it to have a 4π field of view. Both instruments together will have a nominal mass of 6 kg and will need a total power of 12 W. Ions can be detected in a range of 0.05 - 20 keV/q.

Telescope Design: The system has a nominal FoV of 360° in azimuth direction and 3° in elevation. To increase that, a field of view deflector

system (FDS) is applied at the entrance (marked as (2) in Figure 9). By applying high voltage to either side of the FDS, the incoming plasma ions can be redirected into the analysable direction. By changing the applied voltage the field of view can be scanned. The maximal angle is up to $\pm 45°$ for ion energies below 10keV and $\pm 22°$ for energy levels above that.

After being steered into a detectable elevation direction, the ion plasma enters the Variable Geometric Factor System (VGFS) indicated as (1) (Figure 9). If a voltage is applied to the inner electrode, that reduces the overall bandpass, which allows a variation of the instrument's sensitivity. This is necessary since the instrument encounters very different plasma populations during one orbit.

Now the ions enter the actual energy analyser, indicated as (3) in Figure 9. It consists of two hemispheres, which generates an electric field in the gap between these two, if a negative voltage is applied to one of them. The charged particles are deflected, by the field and follow the path of the hemisphere until they hit the position sensitive detector (4). The azimuth direction is determined by the position at which they hit the detector. The highest achievable resolution is 7.5° (Raab et al., 2016).

The LIA detectors are designed to work simultaneously during the entire orbit. Two different observational modes are available. The first one is the normal mode, which will give a full 3D-moment of the plasma every 2 seconds. In this mode the sensor will scan 62 energy levels for 8 different elevation angles and acquiring 24 azimuth angles (180°(per instrument entrance) / 7.5° (accuracy) = 24) simultaneously for each sample. The second mode is called the Burst mode, which reduces the time between two measurements from 2 to 0.5 seconds. However, it will reduce the scanned energy levels from 62 to 30, as well as reducing the elevation steps and azimuth resolution (Raab et al., 2016). This mode will mostly be used at apogee.

1.3 Scientific Goals

The scientific goal is to get a better understanding of the interaction between the solar wind plasma and the embedded interplanetary magnetic field with Earth's magnetic field. This includes observing the ever changing shape and getting more information about the global magnetic reconnection process. Earlier space missions have shown, by in-situ observations, that Earth's magnetic field is compressed by the solar wind on the day-side. However, on the night-side it is dragged outwards into a long magnetic tail (Figure 10). The shocked (bow shock) solar wind plasma is diverted and can only get close and into the ionosphere at the cusps. The shape and position of the interaction isn't constant since the particle flow is changing. It can also dras-

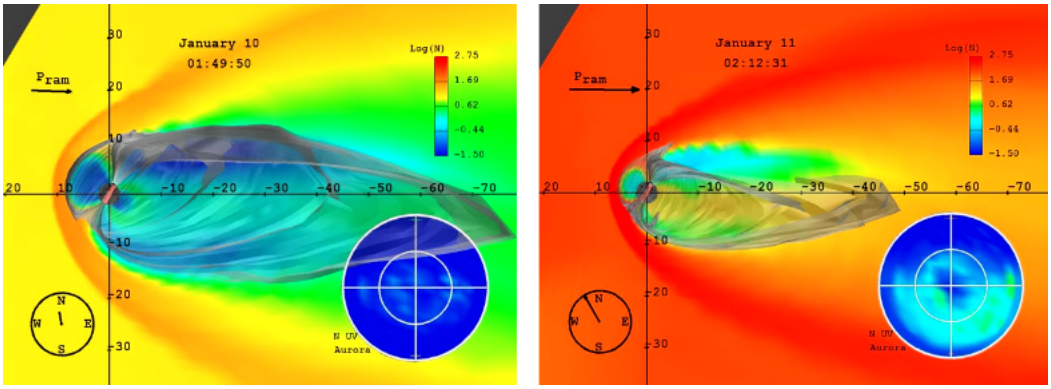

Figure 10: Earth's magnetic field with different solar wind pressures. The plasma density is colour coded, bottom right always shows the aurora. Left: regular pressure. Right: During a geomagnetic storm (Branduardi-Raymont, 2018).

tically change during coronal mass ejections (CMEs), which can be viewed as a large burst of plasma. As seen in Figure 10, a CME, which is also known as a geomagnetic storm, can compress the magnetic field to half its original size. The more plasma strikes the ionosphere, the larger the auroras will be. The shape will be observable with the SXI instrument due to the SWCX process described in Section 1.2.1.

Different processes have been proposed to be the main interaction process between the magnetized solar wind and the terrestrial magnetic field as well as the entry, storage and release of solar wind mass, energy and momentum from the magnetosphere. Relevant processes include the Kelvin-Helmholtz instability on the magnetopause, magnetic reconnection, solar wind pressure variations battering the magnetosphere and diffusion driven by wave-particle interactions. It was found that during southward orientated interplanetary magnetic filed (IMF) periods (solar cycle changes orientation every 11 years), the interaction is increased compared to northward orientation. This was only predicted by magnetic reconnection and is therefore thought to be the main interaction process and was first proposed by Dungey in (1961)

The so called Dungey circle begins when the previously closed terrestrial magnetic flux (Figure 11, panel (A)) are opened as in panel (B). The open field lines are pushed back by the solar wind flow and form the large magnetic tale (panel (C)). During this dayside reconnection opened magnetic flux is now stored as magnetic energy in the magnetotail lobes. Eventually, night-side reconnection closes the magnetic flux which is explosively released and brought back to earth (panel (D)), that results in dramatic auroral displays

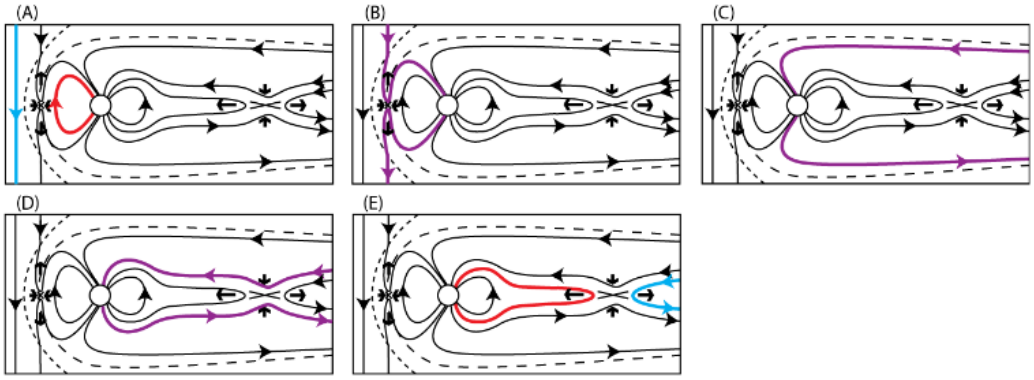

Figure 11: A schematic representation of the Dungey cycle. (Dungey, 1961).

at high latitudes (panel (E)). The three stages (C), (D) and (E) are often known as the phases of the magnetic substorm. Where the energy is first stored in the magnetotail (growth phase), then it is released (release phase) and afterwards everything returns to the normal state (recovery phase).

However, reconnection is a microscale process with macroscale effects and to fully understand it, a bunch of in-situ instruments have been launched, such as ISEE-1/2, AMPTE, Cluster, THEMIS, Van Allen Probes and MMS. These missions confirm the presence of reconnection, but they can not distinguish if it is global or patchy, continuous or transient, caused by solar wind features or by intrinsic layer instabilities. Furthermore, the global rate of opening and closing of the magnetic flux can not be measured. This leads to the possibility of either sending an enormous amount of in-situ measuring instruments to all important positions or doing a global observation.

By using soft X-ray images (SXI, Section 1.2.1), the boundary of the magnetosphere and the location of the cusps can be determined. During the starting and ending phases of a geomagnetic storm, the location and motion of the magnetopause provides information of the amount of closed magnetic flux within the dayside magnetosphere and the steadiness of the magnetic reconnection. The same can be done by tracking the motion of the cusp location, in these phases. Therefore, if both (motion of cusp location and magnetopause) are tracked, a more definite statement can be made.

The observation of the global auroral oval in the ultra-violet range (UVI instrument, Section 1.2.2) is a perfect complement to the soft X-ray images. The size of the oval indicate the open magnetic flux in the Earth's magnetotail. The dayside motion of the auroral oval is another indicator for the dayside reconnection rate, whereas the nightside motion indicates the magnetotail reconnection rate. Furthermore, the nigthside motion can be used to measure the beginning of the substorm, determine the extent of the recon-

nection line in the magnetotail and distinguish between steady and bursty modes of reconnection (Branduardi-Raymont, 2018).

Finally, it is of course very important to know the main properties of the solar wind at the moment of the UV and soft X-ray measurements. This determines if the solar wind changes trigger magnetospheric events.

1.3.1 What are the fundamental modes of the dayside solar wind/ magnetosphere interaction?

Magnetic reconnection can occur in different forms, either it is steady and persists for a long time or it is patchy and time dependent. Which one is present is believed to correlate with the beta of the solar wind (ratio of plasma pressure to magnetic field pressure). For low beta (high magnetic field pressure) steady reconnection is anticipated whereas unsteady is for high beta.

Studying the magnetic cusps is very important to understand the whole interaction between the two magnetic fields. This is because it is the only place where solar wind particles have direct access to lower altitudes. They are factually the region that separates the closed dayside magnetic flux, from the flux that extends far out into the magnetotail. It can therefore give a lot of information about the interaction as opposed to any other region (Cargill et al., 2005). Especially their latitude location provides information about the amount of recently opened magnetic flux. For southward orientated IMF, closed magnetic flux is opened at the dayside and transferred to the nightside. This leads to an expanse of the cusp region in equatorial direction. In the case of northward IMF orientation, the cusps move to the poles. A possible explanation might be that dayside reconnection stops, whereas reconnection on the nightside continues and is transferred back to the dayside (Milan et al., 2003).

In other words, if the amount of open magnetic flux increases, the cusps move in equatorial direction and if it is decreasing, they move polewards. Therefore the location is a direct indicator of the amount of open flux, and furthermore, it gives an idea of the amount of transferred energy with the reconnection process. This is very difficult to estimate with only in-situ measurements available, because it occurs over a large area. Directly related is of course the dayside and nightside reconnection rate, which can therefore as well be derived out of the cusps latitude positions.

Additionally, the solar wind dynamic pressure may also cause the cusps to move, whereas an increased pressure would lead to an equatorial motion and a reduced movement in opposite direction. This was predicted after several simulations and different observations from instruments in the LEO

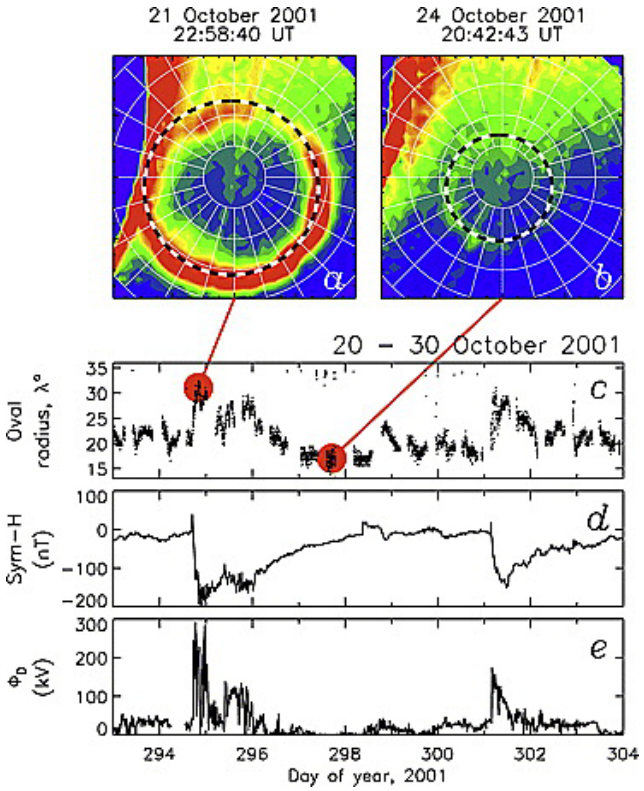

Figure 12: The identification of the auroral oval, at the height of geomagnetic storm (a) and during a phase of quiescence (b), time dependent plot of the polar cap size (c), the ring's current intensity (d) and an approximation of the dayside reconnection rate derived from solar wind conditions (e) (Milan, 2009)

(low earth orbit) e.g. DMSP. (Pitout et al., 2006, Palmroth et al., 2001)

The last effect that influences the cusp's locations is the east-west or also called dawn-dusk component of the IMF. This provides an additional motion, which is not in latitude, but in longitudinal direction. In contrast, dawnward IMF orientation causes the northern cusp to move dawnwards and the southern cusp duskwards. For duskward orientation the directions switch (Taguchi et al., 2009).

1.3.2 What defines the substorm cycle?

The polar cap is defined as the region inside the auroral oval (shown in Figure 12), with open magnetic field lines. The size of the polar cap changes directly with the amount of open flux stored in the magnetotail lobes (e.g. Milan et al., 2012). Therefore, it is a good indicator of the reconnection rate at the dayside as well as at the nightside.

The substorm is in general well understood, but a controversial issue remains of how it is started. Is it externally triggered by a change in the IMF or the solar wind plasma, or is it onset spontaneously as a result of internal processes? How large the external changes would have to be is unknown.

Arguments for both sides can and have been made in the past, however the starting point has still not been confirmed (Hsu and McPherron, 2002, Boudouridis et al., 2003, Huang, 2002). Another viewpoint is that the external solar wind condition provides only the general configuration of the magnetosphere for substorm expansion onset. When and where it occurs depends on the ionospheric conditions as well as internal local magnetospheric parameters.

Another aspect that has to be taken into account are magnetospheric saw-tooth events. These are large-amplitude, quasiperiodic (i.e., not strictly periodic) oscillations of the energetic particle fluxes with an amplitude of 2-4 hours. During such events, the auroral oval contracts and expands within a period of hours. It is not fully clear if they are caused by quasiperiodic substorms or by some different global disturbances (Henderson et al., 2006).

However, not only the size and intensity of the auroral oval are important, but also low-intensity features called auroral beds (Figure 13). They develop in pre-breakup auroral arcs and eventually produce the initial brightening and substorm expansion onset (Henderson, 1994). Auroral beds differ from case to case, as they have a specific wavelength and corresponding exponential growth in the auroral intensity. These are defined by the state of the magnetosphere just prior to a substorm onset. Auroral beds are caused by two main processes: cross-field current plasma instability and kinetic Alfvén waves (Kalmoni et al., 2018). To test these instabilities further observations from the ground in combination with data from SMILE will be ideal.

1.3.3 How do coronal mass ejection-driven storms arise and what is their relationship to substorms?

Coronal mass ejections (CMEs) are mass ejections from the sun's corona which propagate through space at the same speed as the solar wind. They can contain intervals of southward oriented IMF, which leads to more magnetic reconnection in the Earth's dayside magnetic field and therefore to geomagnetic storms. The strength of the storm depends on the length of the southward IMF interval, the field strength and the flow speed. In general, CMEs can be associated with the largest geomagnetic disturbances. Since CMEs aren't a rarity (of course varying highly in intensity), geomagnetic storms occur on a daily basis (Gonzalez et al., 1999).

However, a lot of questions still need to be answered: Is the only deciding factor of a storm occurrence the solar wind and its properties? What is the relationship between the storm and substorm? How are geomagnetic storms and substorms related? Moreover, the start of a storm has always been the most important question for scientists, but the importance of the

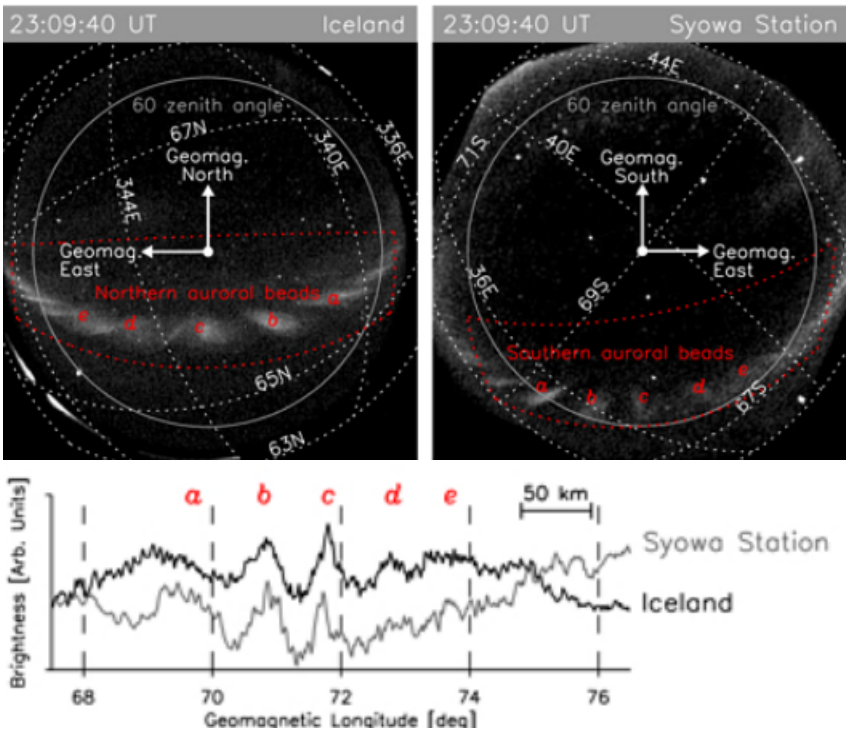

Figure 13: Top: Picture of auroral beds over the north and south pole, Bottom: Comparison of the brightness of the auroral beds between north and south observation (Motoba et al., 2012).

duration of geomagnetic storms is raising, mostly to tell when extreme space weather events will be over. This would give an indication when the danger for technical equipment in space and on the ground has departed. And what is the main reason for a storm to end? Is it that the stored energy in the magnetotail is exhausted or does it stop when the solar wind conditions change?

2 EGSE

Before a satellite can be launched, a lot of development work has to be done. The instruments, the overall design, the hardware, but also the satellites software has to be developed. For the software the same is true as for all other parts: It has to be tested. This is where the Electronic Ground Support Equipment (EGSE) comes into play. It gives the user a possibility to communicate with the spacecraft and the software in particular. It is also individually created for every mission and can have different tasks depending

on what is needed. Here are the main ones:

- Send Telecommands
- Receiving, handling, interpretation, displaying and storing of Telemetry Data
- Power supply of the spacecraft
- Testing of the satellite and satellite parts
- Simulators for ground testing
- Graphical User Interface

Depending on the mission and on the specific requirements, an EGSE may be used to do all this tasks or only a few of those. These leads to the necessity to develop a new EGSE (depending on the differences between projects, a portion of the code may be reusable) for every project. This is based on legitimate motivations, since the projects can have varying main bus standards, testing methods, decoding differences, etc. The EGSEs developed for CHEOPS and SMILE, which will be discussed in Sections 3 and 4, have mostly the same core and the same purpose, to test and communicate with the spacecraft's software. Their main components are the Central Checkout System (CCS), the Test Software Tool (TST) and different simulators. The TST is used to create and edit different test procedures and the CCS is used for the communication with the satellite software. It is however important to emphasize that the EGSE is not used during the operational phases, such as the launch or during the flight. This is done by the Mission Control System (MCS), even if there are many similarities such as the Telemetry Data Monitoring and the telecommanding. For the specific missions discussed in this work, the SCOS-2000 software (Satellite Control and Operation System) will be used as the MCS. This software developed by ESA tries to solve the major problem of newly developing a whole MCS for every mission. A well tested core was developed, that can be reused for every mission and only a few parts have to be changed. It should also be noted, that the MCS contains more functions such as ground station interfaces, navigation, a large multi-user support and it has to be in a very controlled environment with a lot of safety operations, to secure the operation of the satellite as it is in flight. One may say that the MCS is the more powerful tool even if there are some similarities.

3 CCS-CHEOPS

3.1 Introduction - CHEOPS mission

The Characterising Exoplanet Satellite (CHEOPS, Figure 14) is the first S-class mission of ESA's Science program. S-class missions are a lot smaller compared to M- or L- class missions as the development phase should not exceed 5 years and the cost for the ESA Science program is capped at 50 M€. Its main goal is to determine the radii of previously known planets orbiting around bright stars, using ultrahigh precision photometry. All these planets have been found by high-precision Doppler spectroscopic surveys and therefore their masses are known. With the radii measurements of CHEOPS, the density can be calculated, as a result one can make a good assumption on how the planet might look like. With CHEOPS, exoplanets will be identified that have a high potential and are prime targets for longer follow-up observations with ground based instruments as well as with future space missions.

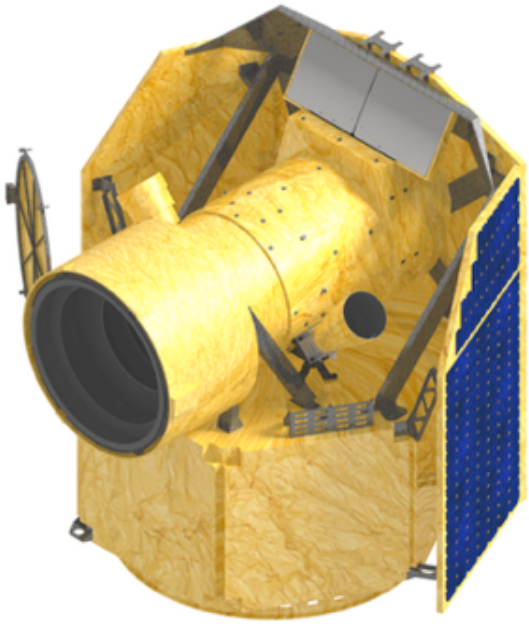

Figure 14: Artwork of the CHEOPS satellite. (*Alchetron Website*)

The main targets are exoplanets with masses starting at slightly above the Earth mass up to a Neptune mass. The radii measurements will be made with a never before reached precision of 10%. This will shed light on a few unanswered scientific questions, like on planet evolution and their

respective paths. For example, it isn't entirely clear why Uranus and Neptune weren't able to aggregate more gas around their large solid cores. Even the formation of these large cores is a challenge beyond 10 AU. This leads to the assumption that these planets formed at different locations than where they are today. Observing exoplanetary systems and especially Neptune-like planets at different times during their evolution will give us a better understanding.

Also with the precise measurements it will be possible to determine the type of these exoplanets. It will be feasible not only to differentiate between gas giants and small rocky planets, but also to identify ocean planets and super Earths with large H/He atmospheres. Furthermore, there will be a first observational estimation on how large a rocky core has to be that it can aggregate gas in a run away fashion.

80% of the observational targets have already been determined by the CHEOPS science team, the rest is still chosen by the astronomic community. With the large fraction of the sky that CHEOPS can get to, a large amount of options are possible. With the sunshield always pointing towards the sun, it will be possible to do observation during more than 80% of one orbit.

3.2 Introduction - CHEOPS CCS

At the beginning of the development phase of the CHEOPS mission, it was searched for an easy and user-friendly way to communicate with the instrument flight software (IFSW). It was first looked at commercial products, but those can't be individually equipped. Additionally they offer a lot of functionalities that aren't needed for the project, which makes them a little harder to use and which are, on top of that, very expensive. Therefore it was decided to develop a specific software tool that is optimized for the project. The main goals were always to get a nice visualisation of the incoming/outgoing data (packets) as well as controlling the most important functions graphically, but at the same time having complete control over the whole script from one place. At first it was decided that Python would be the main programming language that would be used as it is very user friendly, which is important to make the Central Checkout System (CCS) easily interactively controllable at all times. Furthermore, C code can be implemented in Python if it was ever needed to increase the computation rate and it is a widely used language where one can be sure, that it will be updated for the foreseeable future. For the graphic implementation it was a close decision between Qt and Gtk, where in the end Gtk was chosen, because it is widely used on Linux and seamed a bit easier and more natural to use.

After the main decisions have been made, development started with the

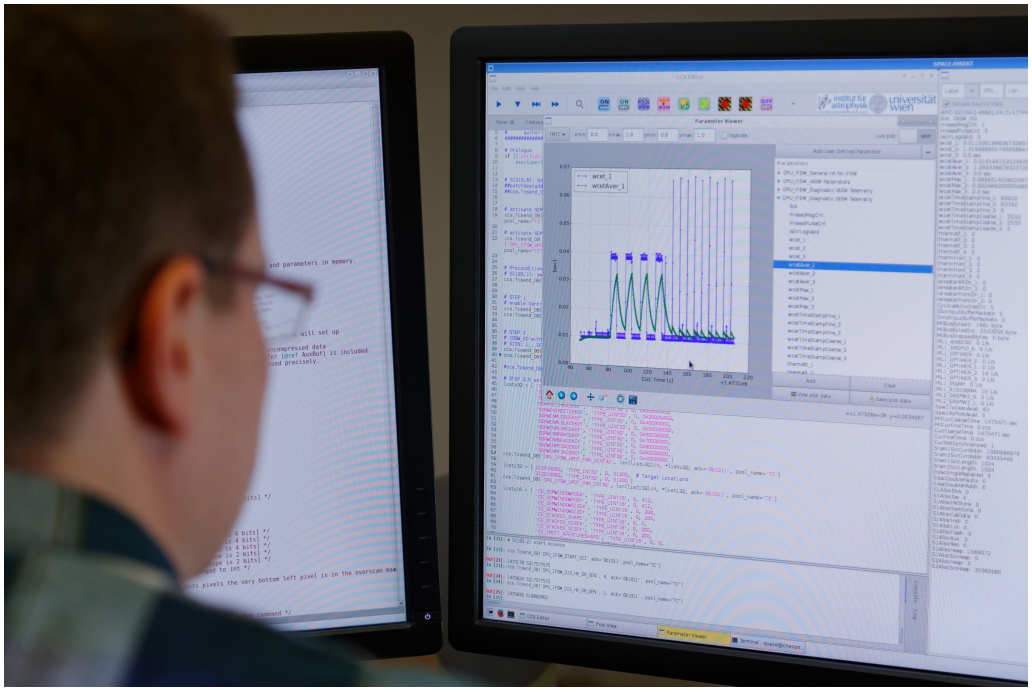

Figure 15: The CCS, how it is typically used during development and testing phases (Kerschbaum, 2020).

editor (Section 3.3.1), which was created first as it contains the IPython shell from where the rest of the components will be started. Shortly afterwards, the pool viewer was built as it is the main tool to visualize the data. This was the first time the CCS could be used to a certain extent, but quickly another problem arose. All the data were stored in the pool viewer, which is faster for a limited amount of data, but for the pools with millions of packets the used machine will get to its limits. If this happens, the pool viewer crashes and the data are lost. As a solution a Database (DB) was created and with it the pool manager to store the incoming packets there. For the DB, SQLAlchemy was used. SQLite was also in the conversation as it is faster, but it can have some problems with multiple accesses at the same time, which was absolutely necessary as all parts of the CCS want to read from there. Additionally, problems with a large amount of data can occur. After SQLAlchemy was selected it was also used to access the IDB, which is necessary for the building and then sending of TC packets as well as the decoding of the TM packets. The IDB for CHEOPS is externally generated by DABYS, however for SMILE it is done internally.

Continuing with the development of the packet manager allowed building of TC packets and decoding TMs. At this point the core of the CCS is

generally finished. However, a lot of time was still invested into perfecting the core functions as well as writing new ones that either increase the usability or add additional functionalities (e.g. the parameter monitor).

The general explanation on how to set up the CCS and information about the first steps can be found in the "CCS User Manual" (Mecina and Ottensamer, 2018). More detailed information on the individual components, what can be done with them and how they work can be found in Section 3.3.

Nowadays, the in-house developed CCS is one of the main EGSE tools used for flight software development and testing (Figure 15).

3.3 Components

The CCS for CHEOPS consists of five main parts, all of them have different tasks and are in separated files:

- *Editor:* (editor.py) Is the main window, CCS can be completely controlled from here, mostly used to write and execute scripts.
- *Pool manager:* (pus_datapool.py) Manages the connections to the satellite software and stores, receives and sends data, however doesn't have a GUI.
- *Packet manager:* (packets.py) Does the decoding and acts as a function library, however doesn't have a GUI.
- *Pool viewer:* (poolview_sql.py) Reads from database and displays the received and sent packages.
- *Parameter monitor:* (monitor.py) Monitors individual parameters and warns the user if something isn't right.

Even if those are the main parts, they are all included in the same program and run in the same instance. This has the advantage that they share the same namespace, and therefore every function in all parts can be called from the IPython terminal integrated in the editor and they can easily communicate with each other by sharing the running instances. On the other hand, if one has a failure and crashes, the whole program crashes. Additionally opening multiple windows (instances) of the same part isn't supported.

In the next chapter, a closer look is taken on all parts individually, however please note that this is not an introduction on how to use this program, but it will be shown what can be done and why it was implemented and why it was implemented the way it is. If an introduction is needed to the CHEOPS-CCS please read the "CCS User Manual" by Mecina and Ottensamer.

3.3.1 Editor

The editor is the main window of the CCS, which opens after the program has been started by executing the ccs.py file. It consists of a menu- and a toolbar at the top, a text window in the center and a IPython terminal at the bottom, as it can be seen in Figure 16. In general, it was created to make the use of previously written scripts possible, but at the same time having interactive control over the whole CCS at all times.

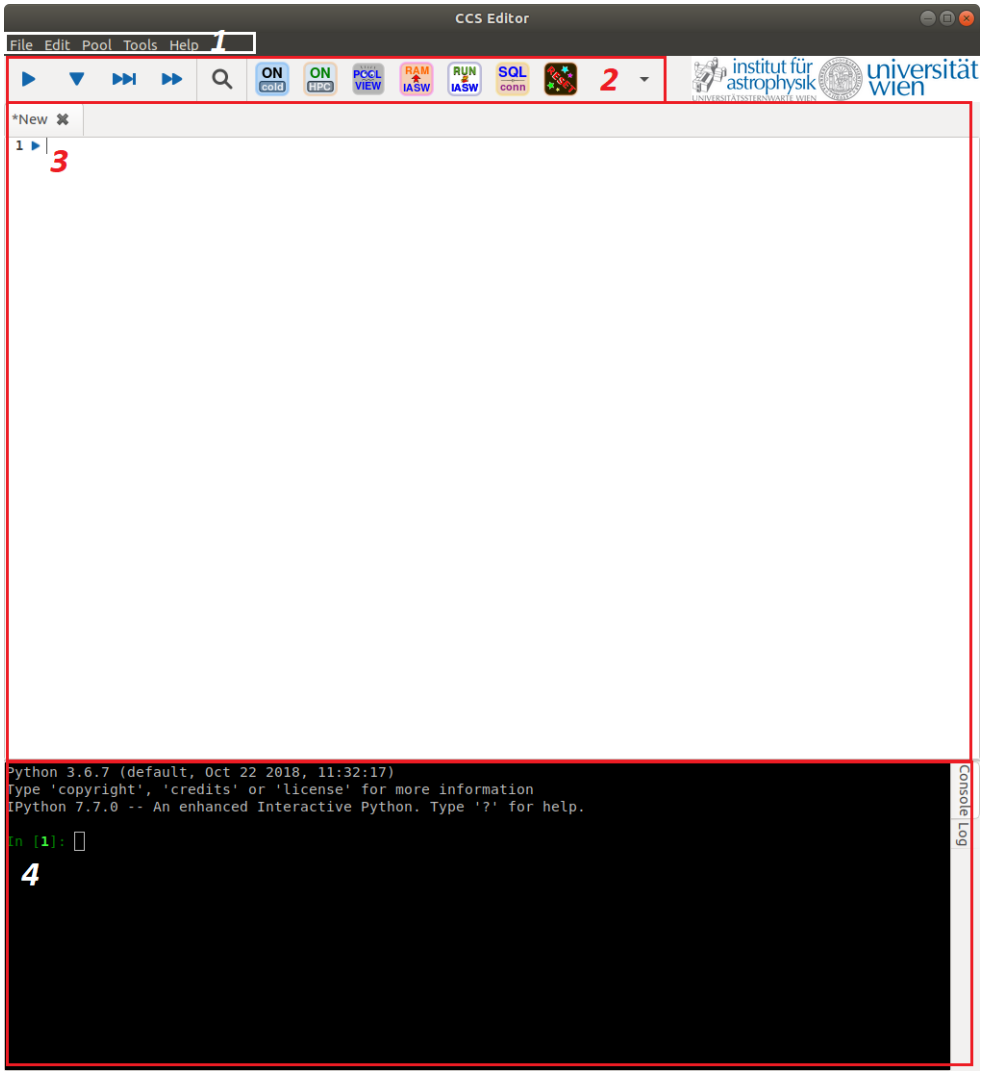

Figure 16: The CCS main window, called the editor, part 1: The menubar, part 2: The toolbar, part 3: The text window, part 4: The IPython terminal (Möslinger, 2020).

The IPython terminal is implemented into the Gtk environment via a VTE terminal, which is a widely used library and the easiest way to implement terminals in Gtk. Its purpose is to start and run all parts and make it therefore possible to communicate with them. With this the CCS gets a central point from which it can be completely controlled. Every function can be called from here and even the shape and colours of the shown windows could be changed. However, this does not include the editor since it is obviously not started in the terminal and shares therefore not the namespace with all the other parts. It can not communicate with any other part of the CCS, however it has no need to do it. Nonetheless, the terminal is the main part of the CCS from where all other parts are controlled and where the user can interact, which is a huge advantage. Interestingly, the central point is both an advantage and disadvantage because all parts run in the same instance and out of the same IPython terminal a crash of one of the parts leads to the crash of the entire instance. Generally, a crash should and will not happen, however, it cannot be completely eliminated. Especially during development where errors and crashes are more likely, it can be very challenging and annoying. Every change to the software has to be tested, but if a change is made to a different part than the editor, it still has to be started every time.

When the terminal is initialized, it automatically executes two files (.ipyloadcfg.py, .ipycfg.py), which imports the most used libraries and as a result, the user can interact right away and doesn't have to bother with that.

Another disadvantage of running everything out of a single python terminal, is that commands have to be typed in every time the CCS is started. Therefore, the script window was developed and inserted just above the terminal into the editor window. This allows the user to create, load and execute Python scripts. Specific command orders, which are used a lot of times, can now be saved. An already created script can be loaded in the "File" layer in the top left corner or, when ccs.py is executed, a file can be given as an additional argument, which will then be automatically shown after the start. The python script startpv.py, which contains the most used commands, such as starting the pool manager and pool viewer (detailed information about these parts in Sections 3.3.2 and 3.3.4), as well as starting a connection to the OBC, is given in the CCS folder. To execute the scripts the user can use the blue "play" buttons at the left side of the toolbar. It is either possible to execute the whole script or only one line. To get the commands to the terminal, a function reads the lines, and uses a built in Gtk function to send the strings. One has to be careful that the command line is empty, otherwise the new command will simply be added to the end of the existing command and then be executed, which of course leads to a nonsense command and most certainly to an error.

Loading scripts and executing them can be to circuitous when only a few commands are often needed. Therefore, the editor contains Action Buttons, where a series of commands can be executed with a simple click. These Buttons are shown in the toolbar on the right side and can individually be assigned to python scripts, so for simplicity users can create a CCS optimized to the personal needs. Per default, all buttons are empty after the installation.

Lastly, the editor also initializes the Logger for all CCS parts. This is done by setting up a TCP-server in a seperate thread to which every other part connects and sends there logging messages to. It stores all logs into the same file, that will always be renewed if the CCS is restarted. All files are stored in the 'Logs' folder and are named with the time and date of their creation. However, it is also possible to view the logs in the editor (log viewer). It is hidden "behind" the terminal since this is only a layer which can be switched on the right side. The Logger is integrated into the editor and passed along to all parts. Its use is to not only write to the Log File, but also to the log viewer, so that all log entries can be seen there.

3.3.2 Pool manager

The next big part of the CCS is the pool manager. It is often not recognized by the user, since it does not have a GUI. However, it is indispensable for the whole program. Its main tasks are to manage the connections to the OBC as well as sending and receiving packets.

The whole communication between the OBC and the CCS is defined by the SCOS 2000 standard. It defines different packets that are built up according to the Package Utilisation Standard (PUS). The two main types are the Telemetry (TM) and Telecommand (TC) packets. Each of them has a huge amount of different subtypes, but they all consist of the same primary header, an either TM or TC secondary header and a data part. All of them can be identified by a service and a subservice type. Each one is used for a slightly different task, but in general, one can say that TM packets get data from the OBC to the CCS and TCs contain commands which are sent from the CCS to the OBC ("SCOS 2000 User Manual", *ESA Website/SCOS-2000*).

The exchange of information is realized over two sockets. One is only to receive TM packets (nothing can be sent), and the other sends the Telecommands and receives acknowledgements (TMs) that the command was received, a progress or error report and a finalization. For every TM connection a new thread is opened, which at all times, listens to incoming bytestrings, that can be interpreted as a packet. For every packet the header is decoded

and stored in a SQL database, as well as the additional data. These data can be decoded by the package manager and be shown in the pool viewer (Section 3.3.4), but is only done if commanded by the user. The same thing is done for the TC socket. If a TC is sent it always listens for incoming acknowledgement TMs to decode and store. Both sockets can be set up with commands in the IPython terminal, they can as well be found in the previously (Section 3.3.1) discussed startpv.py file.

During the communication a lot of packets are sent and received. If there are multiple connections at the same time or simply after a few connections have been made, it is very easy to lose sight which packet came from which connection. Therefore all packets received from the same connection are saved in a pool. To be able to assign a packet to a pool after it was stored, every pool gets a certain identifier which is attached to every stored packet.

It is important to note, that only if both the TC and TM socket get the same 'pool name', they will be saved and shown in the same pool. However, the user should be careful, if a 'pool name' is used again for a TM connection, the pool manager will delete the first pool and all its entries into the database. Normally, the user saves the important pools and test results to files via the pool viewer (Section 3.3.4) or the user has a specific name for every pool and test, such as a date and time code.

Interacting directly with the database is not possible for the user inside of the CCS, but one can use the scripts in the 'database' folder. Executing the 'tm_db.py' file will set up the database. It is required to do this before the first use, otherwise the CCS does not have a place to store the incoming data and as a result does not work. A SQL schema will be created, that contains a lot of tables. Most of them are used to define the structure of all TMs and TCs according to the SCOS 2000 standard (Section 2). They are not edited by the CCS, but are read to decode TM and build TC packets (Section 3.3.3). The two tables 'tm' and 'tm_pool' are used to store data. The 'tm_pool' table holds the information of which pools have been received. A specific identification number is here linked to the given name. In the 'tm' table all incoming TMs are stored with the addition of their pool identification number. Linking these two tables is necessary to find out, which TM belongs to which pool name.

Furthermore, to configure the database one has to change the config_db.py file first, before setting it up. Most importantly, one has to give the personal SQL user name with the corresponding password, otherwise a connection to the DB will be blocked. Also it is possible to change the schema name, which can be useful if SQL is highly utilized and it is necessary to keep everything nicely organized. The other setting should not be changed, since this would lead to malfunctions in the CCS.

Lastly, to check if the DB set up was successful, the test_db.py file is used. It performs a simple test if all tables exist and will write a test pool/packet to the storage table. All of the interaction can easily be used with the corresponding makefile in the 'database' folder.

3.3.3 Packet manager

The packet manager is, measured by the length of the code, the biggest part of the CCS. Its main tasks are the decoding of the full TM packets to a human readable form, the building of TC packets and acting as a Function Library.

With Function Library is meant that the package manager File contains a large amount of functions that may be useful for multiple parts of the CCS, so that they do not have to be implemented in every part. Functions like this are mostly small ones like the conversion from Hexadecimal to Decimal, some however might be a bit more tricky. To use all of them, the instance of the packet manager has to be given to every other part where it then can be accessed by the variable self.ccs. If the procedure in the startpv.py file is followed then it will also be accessible from the IPython terminal in the editor. However, the Function Library also contains functions which have the only purpose to increase the usability of the CCS. It is also easily expendable for future requests.

As one can imagine the decoding of the full TMs is a very tricky issue. Every package type contains different parameters and has therefore a different length. Besides, some of these do not even have a fixed length and can vary. First of all, the packet manager reads the service and subservice type (to identify which one it is) from the already decoded header and then it checks with the IDB how many and which parameters there are. For this a lot of SQL searching and linking tables is necessary. Afterwards, it decodes the whole TM with the obtained information. This procedure is not done automatically, but is has to be requested from the user in the pool viewer. During some testing procedures it may be useful/necessary to get a specific set of parameters in one TM, which does not already exist in the IDB. If the OBC is adapted that such a packet is sent, the CCS would not be able to decode it. However, in the packet manager it is possible to add a new, user-defined TM. One has to pass every information about the new packet (service, subservice type, all parameters,...) to the add_tm_decoder function (Mecina and Ottensamer, 2018). The new TMs will be stored in the CCS configuration file (egse.cfg) and can still be used if the CCS is restarted.

Building the TC packets has some similarities with the decoding of the TMs. Again the pool manager has to do a large amount of IDB linking and

searching to get the structure of the needed TC. After the information of how to build the specific package has been obtained from the IDB, the pool manager uses the given parameters to make the TC. Every TC is different and needs different parameters. It is the user's task to know which and how many parameters to pass along. For more extensive tests the TST part of the EGSE is in general used, as it makes the building of TC more user-friendly, however it is not discussed in this work.

3.3.4 Pool viewer

Displaying the obtained packets in a human readable form is the main task of the pool viewer. It has the most extensive GUI of all parts in the CCS. Nearly all of the functionalities can be used graphically, which does not mean that one can not control it from the IPython terminal, but that for the most part it is not done this way.

The GUI consists of a large amount of different parts: toolbar, filterbar, packet viewer, info bar and the decoding bar. Furthermore, an additional window to plot single parameters can also be opened. All of these parts will be described in this Section.

At first, we take a look at the toolbar. It has a lot of different buttons and functionalities. Since most of them are used daily, I want to give an overview of what can be done here. Starting on the left, the shown pool name is displayed (Figure 17). Since in Figure 17 a saved pool is shown the name has no 'LIVE' caption next to it, which would indicate that the pool is still receiving packets. The name is either obtained when a previously saved pool is reopened or it has to be given to the viewer by the user. After a connection is made by the pool manager to the OBC, it is not automatically shown in the viewer which is achieved by following the previously mentioned startpv.py file, as it contains the needed commands. It is also possible to open multiple pools in the same viewer, independent of the being live or loaded (static). One can switch by simply clicking on the now shown pool name. The viewer always keeps track of which pools have been loaded and what characteristics they have. However, the viewer only has locally saved the shown packets (more detailed information will be given below, when the packet viewer is described), and therefore has to call the SQL Database if the pool is changed.

Continuing with the toolbar, the first button from the left opens the parameter plotter which will be described further below in this Section.

The next one lets one extract all packages of a specific type (identified by service and subservice type). Those are then saved in the packet manager and can be accessed in the IPython terminal. This may be useful for some

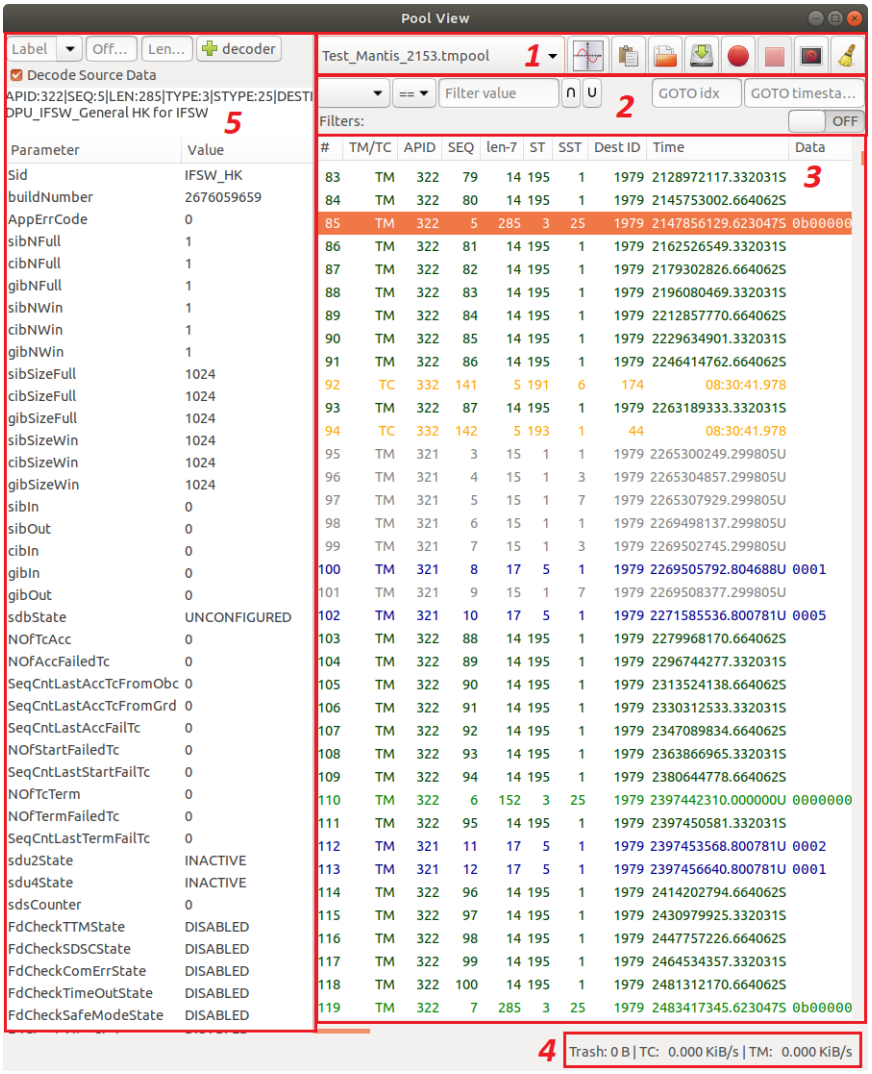

Figure 17: CHEOPS CCS pool viewer, part 1: toolbar, part 2: filterbar, part 3: packet viewer, part 4: info bar, part 5: decoding bar (Möslinger, 2020).

tests, if one wants to check all the received bytes individually.

Going on, the next two buttons have complementary functions. One saves the active pool to a file, the other one loads it again later. If a pool is saved the viewer checks the active pool name and accordingly identifies every packet that belongs in the DB and saves them. If loaded again, it is first checked if the pool already exists in the DB. To be certain to find the right one, the pool name and the modification time (time when the pool was last changed in the DB) have to match. If they do, the pool is loaded from the DB, but if they do not, the data will be loaded from the file and saved into

the DB and then be displayed. If the pool name exists and the modification time does not match, the existing name is overwritten.

The next two buttons have the purpose of giving a small graphical control over the pool manager. The first one calls the function in the pool manager to set up a connection to the OBC and displays the incoming packets. Of course, the user has to specify all needed parameters, in a small pop-up window. The other one aborts an existing connection by calling the corresponding function in the pool manager.

Finally, the last functions of the toolbar are the large data viewer and the pool clearance and those are the rarely used ones. The first one is used if a pool has an huge amount of entries and one wants to get a general overview on what it contains. It graphically shows how many different packet types there are and how often they occur. With the clearing option one can delete all received packages, but continue listening. This is again something, that is implemented into the GUI of the pool viewer, however it calls the deleting function of the pool manager, which removes all entries of the active pool in the database. Afterwards the viewer reloads and only finds the new entries. Clearing the pool is only possible for live pools, since deleting only the entries, but not the entire static pool would not make any sense.

The largest area of the window is consumed by the packet viewer. It shows all packets, that are in the DB and in the shown pool, as well as some information about them (Figure 17). They are also colour coded, to instantly know which type of packet it is. As noted before, only the shown packets are locally stored, the rest is in the DB. This means, scrolling the window is coupled with a SQL query to get the new ones to show. This has the big advantage, that there is almost no difference concerning the amount of entries. Since always the same amount is stored locally and SQL is nearly independent of the data amount for queries that search for specific indexes. On the flip side, a lot of additional functions are needed, to make this possible, because the typical Gtk Functions to set up such scrollable windows do not support SQL queries naturally.

A lot of times one does not want to look at all available packets, but only a few selected ones. Therefore, the filterbar was included. It gives the user the possibility to set one or more filters (it can be decided if both have to fit for a packet or if one is enough) to get only the needed packets. The filter can always be deleted or turned on and off. Filtering is realised by adding the filter to the SQL query. This can increase the time until the new packets are loaded depending on the amount of packets in the pool and how specific the search is. This would not be a big problem if this was only done once, but since it is done always when the window is scrolled, going up and down through the different packets may take a while to load (depending on the

amount of packets in the pool), which the user will notice as a small lag. A small help is that the viewer not only loads the needed packets, but also a small buffer, which is continuously updated in the background. The filterbar has two additional features, that speeds up scrolling. It can either jump to a given index or to a given timestamp. Both are equally implemented (again by adding it to the SQL query) and can be very helpful if the amount of packets is large.

The info bar is the only thing not changeable for the user. It gives some quick information how much data is going in and out per second. The calculation is done by the pool manager, who adds up the size of every incoming and outgoing packet. The viewer gets this information every second and resets the variables to zero. This process runs in an independent thread. In the example window (Figure 17) these numbers are zero since a static pool is shown, therefore there was no incoming data.

The whole left side is dedicated to show the decoded data of packets. It shows all the parameters in the selection and their assigned value. One can either choose to see the decoding in a human readable form like in the example or byte position, the according value in hexadecimal- and decimal system and the bytes. By selecting and deselecting of the box in the top left corner the user can change between these two options. For both the pool viewer does pretty much the same. It first checks which formatting one wants to have, then gets the whole data of the selected packet from the DB and finally sends the data to the packet manager who does the decoding (this is described in Section 3.3.3) and returns the data in the requested format. Similar to the user-defined TMs in Section 3.3.3, the viewer allows the user to define a specific parameter to decode. It can be done by entering a name, the byte position and the parameter length in the top left corner and adding it. Since only hexadecimal, decimal and bytes can be shown, the decoding process is not done by the packet manager, but by the pool viewer. It can be used, if an additional parameter is needed, that exists in a lot of packets. Mostly those are parts of the header, that are not already shown or the user wants the byte values of an existing header parameter. These user-defined parameters are also stored in the CCS configuration file, but have to be added every time the CCS is restarted.

Parameter plotter The plotter has a lot of functionalities and could be viewed as an own part of the CCS, but it is in the same file as the viewer and therefore a part of it. However, it's an own class and has its own GUI, which can be seen in Figure 18. It was created, because the viewer can show all packets and their values, but it is not possible to get a trend of a single

parameter. This can often be a parameter from the housekeeping TMs, which contain the most important values and is sent regularly (every few seconds).

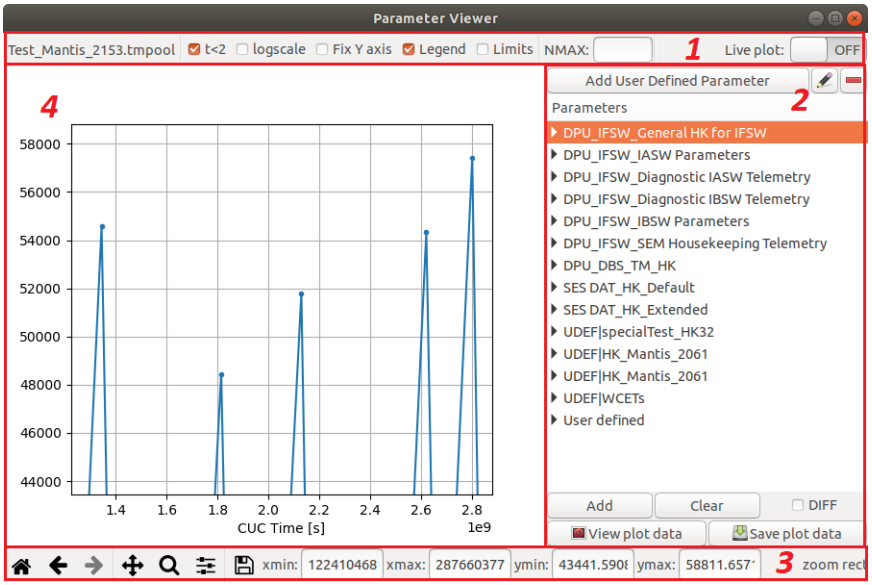

Figure 18: CHEOPS CCS parameter plotter, part 1: settings bar, part 2: parameter bar, part 3: view settings bar, part 4: plot (Möslinger, 2020).

The plotter is exclusively used through the GUI. It consists of the following modules: settings bar, parameter bar, view settings bar and the plot itself.

The main window of the parameter bar is a selection window. It is used to select any parameter of a housekeeping TM, user-defined packet or user-defined parameter, which should be plotted. After the selection was made, it will be shown with the button 'Add'. This will start an SQL survey, that filters the needed rows. The selected ones are then passed to the packet manager which again decodes every single row according to the IDB and returns the needed values. Every parameter by default is plotted against the Time when it was received. However, it is also possible to show it against another parameter, therefore, the DIFF option has to be toggled in the GUI and a second parameter has to be added. The process in the background is widely the same, except that it is plotted on the y- and not on the x-axis. If the DIFF option is not activated and a second parameter is added, the plot will show both parameters against the time. If only the new one should be shown the user has to clear the plot first and add it. The bottom two options are to view the plotted data in a numerical form, which can be useful if a specific value has to be matched to a packet, and the user can save the

numerical data in a file. The last option one has here is to add, edit and remove a user defined plot parameter. This is not the same as in the viewer window since one can specify for which TM this parameter should be used. However, the rest is defined as it was before, with the byte position and the length. Again everything is saved in the CCS configuration file, but in a different Section. One will not see the user-defined parameters from the viewer in the plotter, because they do not hold enough information, but the ones defined in the plotter can be seen in the viewer.

The next part is the settings bar. It allows to adopt the plot to best fit the user's needs. All of these options are a visualisation of Gtk plot commands. Except for the outer left, where the plotted pool name can be seen (Section 18) and the outer right, which allows to switch to a live plot. The plot will be renewed every second so one can follow the trend of a parameter.

On the bottom is the view settings bar, which has a similar use as the settings bar. However, here it is not changed how the plot is made, but how the view is. It is possible to zoom in, to navigate around the view, to adjust the size of the plot window and to save the plot as a picture. Also one can see the exact numbers of what part is shown.

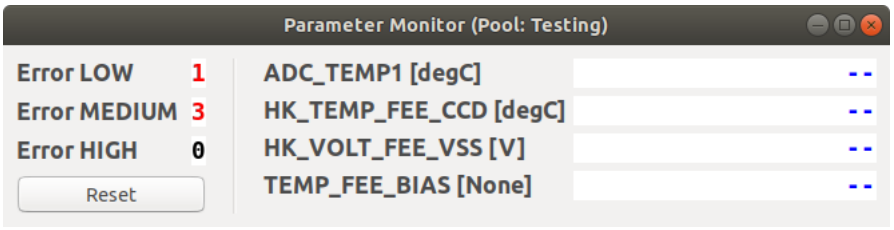

Figure 19: CHEOPS CCS parameter monitor (Möslinger, 2020).

3.3.5 Parameter monitor

The monitor is the smallest part of the CCS. Nonetheless, it is very useful in a lot of situations. It is used to monitor a certain parameter and show if it is still in the acceptable range. Every parameter has two limits which can be found in the IDB. Within the first limit (upper/lower soft limit) everything is perfectly fine. If the values are outside that range, some attention is required on what causes the anomaly, however everything should still work fine. If it reaches the two outer limits (hard limit), the test should be immediately aborted to not risk any damage. Of course if only the software is tested, there is no risk of damaging anything, but at some point the software has to be tested on a Platform Simulator. Furthermore, the monitor summarizes the amount of Errors that happened in a pool and shows them on the left

side. Errors are reported with a specific TM type, which makes it easy to count them, using an SQL query.

4 CCS-SMILE

The CCS for the SMILE mission does look in large portions the same as the CCS for CHEOPS. It was one of the goals to make the transition for every user as easy as possible, therefore a lot of changes will not be noticed even if the process in the background has completely changed. Similar to Section 3 this will not be an introduction of how to use the CCS, but more about the 'invisible' changes that have been made and why they have been made compared to the previous version. Every major requirement will be listed (Section 4.1) and shown how it was satisfied. Furthermore a test script has been created for every improvement. The whole code of every script can be seen in Appendix 1, but a general description of what was done will be given in here. Please note that countless amounts of small upgrades and bug fixes have been done and only the important and bigger ones are noted here.

4.1 Requirements

4.1.1 General Improvements

This Section contains the most drastic changes to the CCS, since they affect all parts.

All parts of the CCS should be independent applications *(#1)*

Description: This may be the biggest change compared to CHEOPS. The parts of the CCS are not started in the IPython terminal in the editor and not sharing the same namespace anymore. Therefore, the applications do not affect each other. This is useful, since in the case of an error and subsequent crash in one application, this would not lead to a crash of the entire CCS. A lot of unsaved changes are not lost anymore and the user does not have to do the whole set-up again. Also, if the terminal is blocked for some reason (sleep command in a script, execution of a consuming task,...) the other parts continue to run, which was not the case for CHEOPS. Furthermore, every application can be individually started. It is not necessary to open the editor first and initialize the pool and packet viewer, before any other application may be started.

On the flip side the applications do not share the namespace, which makes it more difficult to share data and call functions in other applications. The solution is to use D-Bus, which will be discussed in the paragraph right below.

Implementation: To individualize the CCS parts a lot of changes have to be made in the code. Every function has to be checked if any communication with another part would have taken place and if so an individual solution was needed. In some cases the code was adapted, so that it was no longer necessary to call the other one, or some functions have been transferred to different parts of the CCS, like the Function Library (Section 4.1.2). If it was unavoidable to call a function in another application, D-Bus was again used. However, there are some small limitations in doing that, which will also be discussed in more detail below.

Continuing with the individualisation, it was necessary that every application opens the configuration file and connects to the logger, which where previously both passed along by the editor/terminal and all have to connect to the D-Bus. The last thing was to make sure, that if other variables were passed along that they can also be given at the start. Like the name of the Pool that should be opened for the pool viewer and monitor.

Testing: To show that every part is standalone, it is necessary to start it without any of the other parts running. Therefore each part is launched and after a few seconds waiting time it is killed. Afterwards the procedure is redone with the next application.

All applications should exchange data with D-Bus *(#2)*

Description: Even if the applications of the CCS are now independent processes, they still have to exchange data. There are numerous ways to achieve communication between two independent processes. When only small amounts of data have to be shared (like it is in this case), D-Bus (short for Desktop-Bus) demonstrated the best results. It is widely used in GNOME and KDE Linux desktop environments, and therefore, very well documented. Additionally, all applications can communicate with each other, without having to worry about a lot of connection, therefore facilitating the whole process. Multiple implementations such as libdbus, GDBus, QtDbus and sd-bus exist, that allow a connection to the D-Bus. For this project libdbus with the python binding dbus-python was used. The main advantage is that this library is by default installed on every Linux machine. This makes the CCS easier to use, since no additional library has to be installed.

However, it is a low-level implementation and is therefore not for all projects recommended, but in the community it is still the main library, for working with D-Bus. Due to the high usage the terms of libdbus and D-Bus often blur, but it is important that libdbus makes it only possible to use D-Bus.

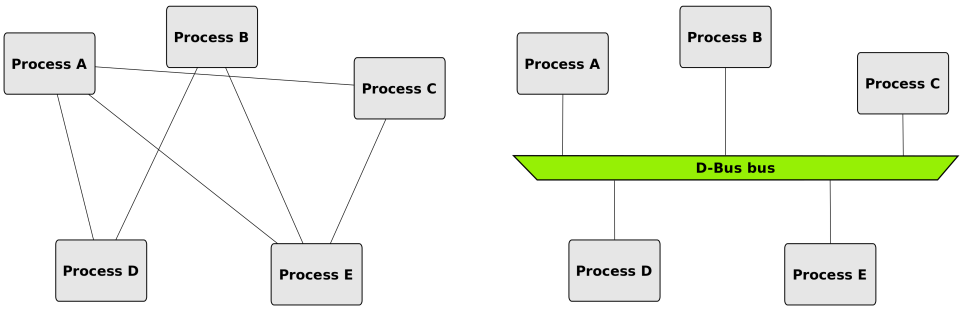

Figure 20: Left: schematic idea of inter-process communication without D-Bus. Right: with D-Bus (Cantero, 2015).

In detail, D-Bus is a software bus, that allows communication between processes, which are concurrently running on the same machine. It was designed to replace the software component communications systems used by GNOME and KDE. The idea is that, if processes want to communicate with each other, it is not necessary to make a connection between them. Instead, it provides a single shared virtual channel that can be used by multiple applications. Every process just connects to D-Bus and can exchange data with every other connected process (for clarity the idea is schematically shown in Figure 20). For a huge number of processes that want to communicate, setting up one-to-one communications between all is no longer necessary. This makes the system more efficient and reliable. If a process connects to D-Bus, it has to give a well known and unique name (bus name), so that it can be indubitably identified. For the CCS all names for the applications are saved in the config file.

Every Linux Desktop environment has multiple D-Busses, one System Bus, available for all users and one Session Bus for every user login session. Both can be accessed by any process. For the CCS only the Session Bus is used, since there is no need to communicate with other users.

Implementation: Calling another process, being the client, is an easy task with dbus-python (libdbus). After the connection to the bus has been made, it is necessary to give the well-known Bus name of the process to call. Then one has access to all of their exported objects. This connection can be

stored in a proxy object. It serves as a stand in for the remote object. A method can now be called on the proxy and if it is exported it will be called via D-Bus. Arguments can be passed along and afterwards D-Bus returns all return values.

Exporting methods, being the server, is a bit more tricky to code, if one is just getting started with D-Bus. However, the basic steps are quite simple. First, one has to connect to the D-Bus, afterwards the specific D-Bus name has to be given. Now that the process can listen to any incoming method calls, it has to be connected to a main loop. Different ones can be chosen, but since a Gtk Main Loop is always present in the CCS, it is linked to the D-Bus calls. To make methods callable for other processes, they have to be exported first. One can either write an interface by hand, which contains the name of all methods that should be exported or use the dbus.service module. Using the module is obviously easier, because one does not have to update the interface every time a new method is created. The module is passed to a class as a metaclass and all included methods will be exported. The connection to the bus and choosing a name has to happen in the init method. However, to open a Gtk window a metaclass is already given in all applications and giving multiple ones is not recommended in Python. Therefore a new class has to be created that will then call all methods over a passed along instance. Since all applications need this, a new file was created (DBus_Basic.py), which contains the code and is called whenever an application is started. This is only one of multiple reasons to create a new file/class for the exporting and connecting process. Others are for example: Problems with Keyword Arguments and specific D-Bus Data Types, which will be explained below. The new file exports three main methods: functions, variables and dictionaries. As the names suggest, it is possible to call all methods (functions inside a class are called method in Python) and get/edit all variable and dictionary values in the passed along class. However, they can no longer be called directly, but the name has to be given as the first argument, as a string. It may not be the most instinctive way to call methods, but similar ideas are already implemented in other python modules. How D-Bus can be used specifically in the CCS is described in the getting-started.py file, which will automatically open, if the ccs.py file is executed. Additionally two more methods are exported: ConnectionCheck and show_functions. The first is a simple function, which can be called to check if the called application is ready to communicate. It will return a small acknowledgment, if called. The second one returns a list of all available methods in the applications. This can be helpful for the user, since method names can no longer be completed with the tab key.

One problem with D-Bus is that it is platform independent and therefore

uses specific data types. Everything that is sent, has to be first converted to a D-Bus type object, with a specific signature that identifies it. The first ones are the Basic types with fixed length (byte → dbus.byte, boolean → dbus.boolean,...), with all the specific Integer types (Int16, Int32,...) having corresponding D-Bus types. There is also the Basic type with variable length that is mostly used for String type objects (string → dbus.String). And lastly D-Bus offers container types which contain other basic type objects (list → dbus.Array, dictionary → dbus.Dictionary, tuple → dbus.Struct). However for these, all the entries have to have the same data type, which has to be given when they are sent. It can be avoided by using Variants. This is a Data type, that contains other basic type objects. Therefore, a container can contain only variants, with different basic types wrapped in them. Most of the time D-Bus will use the right type and the operator does not have to worry about it. But with bigger containers it can be a problem and one has to give the signature to which type it should be converted. The same is of course true for the return values. Nonetheless, those values are not converted back to Python types automatically. This is mostly not a problem since all these types are subtypes of the original Python Types, which means that they can be used in the same way. One exception is the Boolean data type, which can by default not have subtypes in Python. They are therefore converted to a own dbus.Boolean type and are shown in Python as these, with values of 0 and 1. For most of the cases this is irrelevant, but should be kept in mind .

Additionally, non-basic Python data types can not be sent via D-Bus. For example: numpy objects or NamedTuples. Those have to be first converted to basic Python type objects and then be sent. Also it is not possible to send socket objects or instances. These can not be converted to any Basic Python type and there is no other possibility to exchange them via D-Bus.

Furthermore, D-Bus doesn't support None Type arguments, neither in the given nor in the return values. The only exception is if nothing is returned the return will be None. This is because the NULL byte has a specific meaning for D-Bus. All dbus.String objects start and end with it to mark their length and if a connection is made by a client the first thing that has to be sent, is a NULL byte. To make it easier for the user, helping functions have been included in the Central Function Library (CFL, described in Section 4.1.2). They have the same name as the Methods that are exported by every application via the DBus_Basic.py file and can be used similarly, with the only difference, that the first argument has to be the connection that should be called and the second argument is the to calling Method/-Variable/Dictionary. The returned objects will automatically be transformed into Python types to make the output look a bit 'nicer' for the user. Also

None Type arguments will be handled.

Another complication with D-Bus is that it does not pass along key word arguments. There are only key word arguments that are implemented in libdbus, which can be used, but they only affect the method call. For example one could make a call, but does not wait for a response. Therefore, a helping function has been made in the function library, which takes the given keyword arguments and converts them to a Dictionary. This will be passed over D-Bus and then converted back to keyword arguments in the D-Bus basic file. It is automatically done, if the above described functions in the CFL are used.

Testing: To show that information can be exchanged with every application via D-Bus, the application has to be started first. This is done in the same way as in the previous test. Afterwards a connection is made and a check to confirm, that it was successful, is performed. The result is then printed out and the application is closed. For the test a function from the Central Function Library (introduction and explanation in Section 4.1.2) is used, that makes the connection via D-Bus to the given application name.

Still complete control over CCS in the IPython terminal *(#3)*

Description: It was one of the central points during the development of the CHEOPS CCS, that everything can be controlled from one place. This was then realised by running the whole CCS in the IPython terminal embedded in the editor. But in the current version the parts are all independent applications and D-Bus is used. Normally the user would have to make a connection to D-Bus and then save the connection to a specific CCS application in a proxy object, like described above. However, to increase usability all connections will be made automatically. This is done, whenever an application is started. Yet it is not possible to communicate with the terminal directly, because it does not export any methods over D-Bus. Therefore, everything is sent to the editor which will forward it to the terminal. Every connection can then be seen and the created proxy object is ready to be used.

Implementation: All of the applications contain a method (connect_to_all) that connects to the terminal and is called by the DBus_Basic File. It would have also been possible to do the whole process in each application individually, but it was more appealing to give all files kind of a uniform look, where all of them use the same methods when started. Also the editor connects to the terminal, which adds the potential to communicate with it as well, something that was not possible for the CHEOPS-CCS.

Since the editor does not have to be started at first anymore, it is also possible that some applications are already running when the terminal is launched. To still have complete access, the editor has an additional functionality, that scans for other applications and creates the proxy for them. It would have also been possible to put this into the start up files for the terminal (.ipyloadcfg.py, ipycfg.py), but these were wished to be kept as small as possible.

Testing: The test is started by opening each application and when it is running it will send a command to the terminal, via the editor, so that it calls a random method in the started application. It is not necessary to call all methods that exist, since if it works for one it shows that it is possible to call them. The second part is to show that this still works if the editor is not started first, but the terminal still makes the connection. To test it, the same thing is done as before and in the end all applications are closed.

Multiple Instances of one application running at the same Time *(#4)*

Description: For some applications of the CCS, using multiple instances at the same time may not be ever used (for example: editor), but for others (for example: pool viewer) it can be very useful. This may not lead to a wider variety of uses, but to an improved usability. For instance, it can make it a lot easier to compare multiple pools with each other, either in the viewer or in the parameter plotter.

Implementation: The implementation is not as simple as just opening an additional window. First, D-Bus names have to be unique to identify every process flawlessly. Therefore, every application has a number added at the end of the name. To keep track it can be seen at the end of the window name. However, the number leads to the problem that other applications don't know with whom of them they should communicate. To solve this, a dictionary was created in the Central Function Library (Section 4.1.2) which stores the main instance of every application. For every process it is unique and can only be accessed by itself. Every time, if no specific instance is given, the connection will be made to the main instance, which will always be the first one if it is not closed or changed by the user. Nonetheless, if a change occurs, every application is called via D-Bus to adjust their settings. However, it is not only possible to adjust the main instance, but also to change the communication instance for every application individually. This

may be useful in some cases, but the user has to be careful to not lose track of who is communicating with whom. The terminal does keep track of the main instances, it has numbered proxy objects for the user to choose from.

Testing: At first it is shown that two different instances of the same application can run at the same time. The whole procedure is done for every application of the CCS. To show that both are working and are connected the D-Bus a random method is executed in both instances. Afterwards it has to be shown that the main communication can be changed. Therefore the main communication is saved before and after a change was made from Instance 1 to Instance 2 as the main instance. The result is printed out and shows, that everything works. In the end, it is checked if the terminal can also handle it. The editor is started which automatically connects to both and then the terminal calls a method in both instances. In the end all applications are closed again.

Compatibility with CHEOPS *(#5)*

Description: The CHEOPS CCS is not used too often nowadays, since the satellite was already launched in 2019, but if some on-board software updates have to be made it would be nice to test it on-ground with the newest CCS version. Therefore, one can relatively easy switch between the projects, by changing only two things. First of all, the configuration file of the database has to be updated. It has to be assured that the correct DB is used and the correct user has access to it. Secondly, the setting 'project' in the renamed configuration file (now: ccs_main_config.cfg, before: egse.cfg) has to be set either to CHEOPS or SMILE. This defines the used packet structure.

Implementation: To separate these two, the packet structure is no longer defined in the pool manager, but in an additional packet_config file. One exists for CHEOPS and one for SMILE, where the 'project' setting, in the config file, decides which one to use. Everything else can be used just the same for both projects.

By making those additional files it was tried to establish a general core, that can be reused and which minimizes the needed effort for the next version of the CCS, that will be developed for future projects. However, it can never be assured which changes might be needed. If the difference is too big, it could be that a largely new developed CCS might be easier done, than adapting the current one.

Testing: To write one test script for this part is not really possible as settings have to be changed by the user to switch between the two projects. Furthermore, it is not possible to show that all of the existing functionalities work fine for both satellites, due to the huge amount that do exist. A test was nevertheless created, but it was divided into two parts. The first part can be run if the CCS is configured to CHEOPS and the second one for SMILE. It is up to the user to have the correct settings so that the test can be successful. Both parts open the pool viewer and import a previously saved pool with a few thousand received packets. One can see that those are shown and decoded according to the projects packet structure, which is the main difference between those two.

Multiple Instances of whole CCS running at the same Time *(#6)*

Description: To avoid confusion, it will be referred to a whole CCS instance as main instance and to two application windows as two instances. For CHEOPS it is only possible to run one CCS main instance on the same machine, but it may be useful to run multiple ones at the same time. For example running a main instance for the SMILE mission and one for CHEOPS or making multiple complete independent tests at the same time.

Implementation: Every application has to be given a specific main instance name whenever it is started. By default this would be the project it is used for and can be seen at the beginning of each window title. However, a user-defined one can be used, by passing along a string argument during the start (with a hyphen at the beginning and the end of it: "-projectname-").

However, with D-Bus every process can communicate with each other, so to prevent it, a check has to be implemented. It is done whenever an application is started and searches for communication partners, it will as well check, if the main instance matches. If not, no connection is made. Neither is it possible to change the main communication to an application that is not running in the same main instance. Nonetheless, the IPython terminal does not have a main communication dictionary, so it could talk to applications outside of the main instance, but the user would have to make the connection on his/her own (on purpose). A proxy object in the terminal would not be automatically created.

Also, D-Bus names have to stay unique, therefore the numbering of multiple application instances would continue over multiple main instances. There can not be multiple pool viewer 1. Therefore, the main communication would

be with pool viewer 1 for the first main instance and pool viewer 2 for the second.

Testing: First it is necessary to start all applications in two different main instances. Afterwards, it is tried to change the main instance of each application instance to the one running in the other main instance. An error will be raised and the message is printed out. This shows that a communication between them is not possible and thus it is approved that they are running completely independently.

Centralize the Logging process for all applications *(#7)*

Description: For CHEOPS the logging server was set up by the editor and was shut down with it as well. However, it is now no longer necessary to start the editor at first, which made it necessary to create an own script (log_server.py) for the logging process, that can be started from every application.

Implementation: The basic structure hasn't changed much as it is still a TCP Server, where everyone can connect and send the logging messages to. It is now started from DBus_Basic, since this is called by every application that is started. In order to avoid set up of multiple logs, it always checks, if one is already running.

Since the logger should only close if no application is running, it always checks the list of all available D-Bus names and compares them to the saved names in the configuration file. If at least one match is made, it continues to run, if not it will shut down. Also the lowest logging level that should be shown in the file, can be set in the configuration file. As it was for CHEOPS, the name is always a composure of the starting time and date. Additionally, it can be set in the configuration file, how many log files should be maximally kept. If the maximum number is reached, the oldest file is deleted.

Furthermore, the application name, which wrote the log message is still mentioned at the beginning of every entry.

Testing: To be able to show that all applications write the logs into the same file, it is first necessary to produce a logging message. Therefore a similar, but slightly shorter version of the previous test was used which will produce exactly one logging message for every application. Afterwards, the log file is opened and the last entries are printed out, which contain the same logging message of every application.

The UNIVIE Button *(#8)*

Description: At first the simple idea came up to integrate the logo of the University of Vienna (UNIVIE) into the GUIs of every application, to show who the developers are. Shortly after, it was thought to give it some usage, such as simplifying the navigation though the different applications. Now the UNIVIE button opens a drop down menu, which gives the possibility of opening all applications and a graphic way of managing the communication between the individual parts. Changing the communication is important, if multiple instances of the same applications or the whole CCS are running.

Implementation: The whole menu is created in every application, but the executed functions are in the CFL. All the functionalities, that open another application are very short and relatively easily implemented. In the core it is the same as it is done in multiple testing scripts (module subprocess).

Changing the communication this way is a simple Gtk GUI that allows the user to have a better overview what is going on in the background D-Bus communication of the different applications, as well as simplifying the way to change them.

Testing: This is mostly only a graphical upgrade to the previous version and is therefore very difficult to give a test. However, only functions are used that have already been implemented before. Therefore, the test only opens every application individually and just drops down the menu to see that it is there.

Implementation of Different Packet Types than PUS *(#9)*

Description: In a perfect world all of the satellites soft- and hardware would work perfectly from the get go. The CCS would connect to the On-Board Computer (OBC) and does not care about the rest. However, it may be wanted to connect directly to the Sensor Electronics Module (SEM) or the Front-End Electronics (FEE) and skip the OBC. This can either be in the early phases of the development, when not all parts are already working or if just a quick test should be made, for which it is not necessary to start everything. At first, this sounds like an easy task for the CCS, because it does not check to whom it is connecting. If the other parts send packets, everything is fine. The problem is that the communication between the FEE, SEM and BEE (IFSW) is not done with packets that are built according to PUS. For SEM to IFSW, RMAP (Remote Memory Access Protocol) is used

and the FEE has an individual packet structure, which are both wrapped in a SPW protocol. This is not the case for PUS, because the OBC simulator already removes the SPW protocol bits and sends only the pure PUS packet. To communicate with those directly from the CCS one, could either change the FEE and SEM completely, so that PUS is used. However, for the finished satellite, this is not the case and it is therefore not implemented. The other possibility is to expand the CCS, so that it can understand RMAP and FEE as well.

Implementation: If a connection is made, the CCS has to be told which protocol it should expect (either SPW or PUS). To decide if the packet is RMAP or FEE, the pool manager reads the SPW protocol, which contains this information. If nothing is specified, PUS is used, since it will always be the main way to use it. After the connection has been made, the pool manager starts a new thread with a worker, that reads the packets and stores them in the DB. For PUS packets the same worker is still started, but a new one was also created. This one takes the SPW protocol into account and then decodes the RMAP or FEE header and stores the needed parameters into the DB. Since it has to be decided which worker is started it is either possible to have PUS packets in a pool or SPW wrapped RMAP and FEE packets. It is not possible to have both in the same pool, however this is something that is not expected to be happening. The information of the header parameters and header structure was added to the packet_config, which already stored the information for PUS. It is also something that has to be changed for every project, since there might be changes which protocols are used and even if they stay the same, there might be small differences.

Afterwards, the pool manager uses the protocol to decide which table in the DB should be used. There are now not only two tables (tm: stores all the packets, tm_pool: stores all the pools), but two additional ones. The PUS packets are still stored in the "tm" table, RMAP packets in a separate "tm_rmap" and FEE in the "tm_feedata" table. Moreover, an additional column is added to tm_pool, that specifies where the corresponding packets to each pool are. Different tables are necessary, because every header contains different information, that should be stored, therefore different columns are needed. Of course, one could also use one table and leave the not used parameters empty, but to have a better overview, it was decided to use multiple tables.

After everything is stored correctly it is also wanted to view those packets, therefore the pool viewer also had to be updated. If a pool is opened it checks in the database which protocol should be used. If it is PUS everything stays the same, however if it is SPW, there is not only one pool created, but two.

One for the FEE packets and one for RMAP. Both header contain different information, that may be important for the user and the CCS does not know if only one protocol is used or both. Therefore it always creates two tabs, which have an additional label telling the user which protocol is shown. After it was set up, the RMAP table will always be shown first. It also could have been FEE, but it was thought that RMAP would be used more often and therefore it was decided this way.

An important note at the end may be that the Decoding of the Parameters does only work for PUS packets. It was decided, that the time and effort would be too large to include it for FEE and RMAP as well, as they are thought to be only used sporadically. It would have been necessary to create a complete IDB for both. Furthermore, the changes needed in the CFL would have been huge as well. Since the parameters can not be decoded, it is not feasible to integrate SPW protocol into the monitor and plotter. Both could only work with parameters.

Testing: To show that the implementation works, it is necessary to load SPW packets to the CCS. Then it can be seen that the columns will change and the packages are there. First of all the pool viewer and the pool manager are started and first a "normal", previously saved PUS pool is loaded. Afterwards a previously saved RMAP pool is loaded just the same way. After it was sucessfully shown, both applications are closed again.

4.1.2 Central Function Library

Building a Central Function Library without a passed Instance *(#10)*

Description: For CHEOPS a Function Library was included into the packet manager, which will however no longer be used for SMILE. But most of its functionalities have been transferred to the newly created ccs_function_lib.py file, mostly referred to as Central Function Library (CFL). The first of two main reasons to do this major overhaul was to get rid of passing along the instance of the old packet manager. This would have not worked with the creation of independent applications, instead of everything using the same namespace. It is now only necessary to import the CFL to use it. This makes it not only possible to be used by all running applications at the same time, but also by processes that are not part of the CCS, like small working scripts. Of course those could also be loaded into the editor and executed there, but for SMILE this is not necessary and the user has more flexibility. Another important thing to note is, that changes to the CFL made by one

application can not be seen by others. CCS wide variables, therefore, still have to be in the configuration file and not in the CFL. The second reason is much more pragmatic: during the usage a lot of methods (functions inside a class are called methods in python), that were only used for one purpose and later forgotten about, have been added to the CHEOPS Function Library and piled up there.

Implementation: Every method in the CHEOPS CFL was first evaluated and then decided, if it should be transferred to the new ones or not. All of the transferred ones had to be changed, since they should no longer be used in a class instance, but as python functions. Essentially, we want to get rid of the 'self' variable that has to be used in Python classes. Also, if needed, updates have been made to improve functionality, usability or execution time. Furthermore, functions have also been added, mostly those to make it easier for the user to interact with D-Bus and therefore, communicate with all applications. These functions essentially remove the known problems of D-Bus like getting rid of the unaesthetic D-Bus type variables as well as not being able to pass keyword arguments and None Type variables, which was already described in Section 4.1.1. The only difference for the user who is transferring from the CHEOPS-CCS to SMILE is that all functions that have been called – no matter if from a personal script or an application of the CCS – through the variable "ccs" can now be used with the variable "cfl". Of course, every call was updated in the CCS and also it is automatically imported into the IPython terminal.

Testing: This is again an upgrade that involves a huge amount of functions that in general do the same stuff they did for the CHEOPS mission. Therefore, the test shows that the newly implemented CFL works in general how it was supposed to, but not every function in it will be executed and will show that the transfer was successful. The test script imports the CFL (without an instance) and calls a random function to show that it is working.

Change decoding from Bitstring Module to Struct Module *(#11)*

Description: Changing the used decoding does not only affect the CFL, but also the pool manager. However, the pool manager "only" decodes the header of each packet and stores it in the DB, whereas the CFL decodes, if necessary, the incorporated data and builds the TCs. This seems like a bigger and especially more complex task and therefore, this update can be found in this Section.

Bitstring is a Python only module, that makes the creation, manipulation and analysis of binary data as simple and natural as possible and is still updated and refined. It works really well and there is no performance based reason to stop using it. However, it is not by default installed with Python, as opposed to Struct. Since it is wanted that the CCS is as easy to use as possible, all additional modules that are not inevitable or where there is no good alternative, are tried to be replaced. The same reason was also key in determining which D-Bus binding was used.

Implementation: First of all, it was necessary to search for every implementation of bitstring and find an individual solution of how to translate the usage to a struct function. In many cases bitstring and struct offer very similar functions and it was only necessary to change the implementation. In other cases small portions of the code had to be adapted so that it would work with the new module.

Testing: Since the struct module is often used in the two files it is not possible to show that each function works where it is used. Therefore, the amount of usage of both modules is shown. For this purpose the pool manager and CFL files are loaded and searched line-by-line if any module was used, whereas all comment lines are filtered out. The amount of usages is then printed out in the end. As it can be seen, bitstring has completely vanished and struct is heavily used.

4.1.3 Editor

In general the editor pretty much stays the same, there are only a few small improvements and added functionalities, with the most important ones shown here.

Restart only the IPython terminal *(#12)*

Description: Everybody will have the issue every now and then, that a script was quickly written and as a result of some mistake, the code is caught in an endless loop. Normally one would hit CTRL+C to stop the execution and start over, however, this shortcut does not work in the Vte terminal. This would mean, that only closing the whole editor would stop the execution. This is not a good solution, since unsaved changes would be lost. Therefore, a possibility was created to only restart the terminal and not the whole editor. It can be found under the menu layer "Tools/Restart terminal".

Implementation: The Vte terminal will first be removed from Gtk and which automatically closes it. Afterwards, a new terminal is added to Gtk the same way as it would do when the editor is started. All D-Bus connections are made and all standard libraries loaded. At last, the page has to be switched back to the terminal away from the log viewer, which was shown because it was, the only remaining page. A small notification will be shown in the terminal as well as in the Log File.

Testing: First, the terminal has to be caught in an endless loop. Therefore, the editor is started and a small endless loop is sent in via D-Bus. After it is caught, the restart function in the editor is called. This results in a new terminal being set up and everything will work fine again. In the end the editor is closed again.

Communication with the IPython terminal *(#13)*

Description: This is one of the smallest upgrades covered in this work, but it solves an important problem and will therefore be shown.

Normally, the communication of the editor with the terminal is pretty straightforward, as it has a built-in function, that sends the command as a string to the terminal. All other applications can either call a function in the editor via D-Bus to send a command or connect to a socket that is offered also by the editor and send the command there, from where it will also be forwarded to the terminal. However, a problem occurs if the line in the terminal is not empty. The new command will be added to the end of the written line and executed as a whole. This will in most cases cause an error and reduces the usability.

Implementation: A check has been added, that makes sure that the last line is empty. If not, the line is temporarily saved and deleted from the terminal. Afterwards, the new command is executed and the other one is sent in again, but is not executed. The user only sees the execution of the sent in command, but otherwise should not notice any difference. Since, the independent application sends far more commands than it did in the CHEOPS version (e.g. open a D-Bus connection when an application is started), it was necessary to add.

Testing: At the very beginning the editor has to be started, then a command is sent that is written into the terminal, but is not executed. After-

wards, another command is sent that should be executed. In the end one can see that the second command was executed in the terminal and the first one is still there right below.

4.1.4 Pool manager

Pool manager GUI *(#14)*

Description: An average user of the CHEOPS CCS might not even have noticed the pool manager since it was only running in the background and a GUI did not exist. However, this made it difficult to keep track of all the different pools that have been opened and a central point was needed from where a connection could graphically be made. Therefore a small GUI was created, which can be seen in Figure 21.

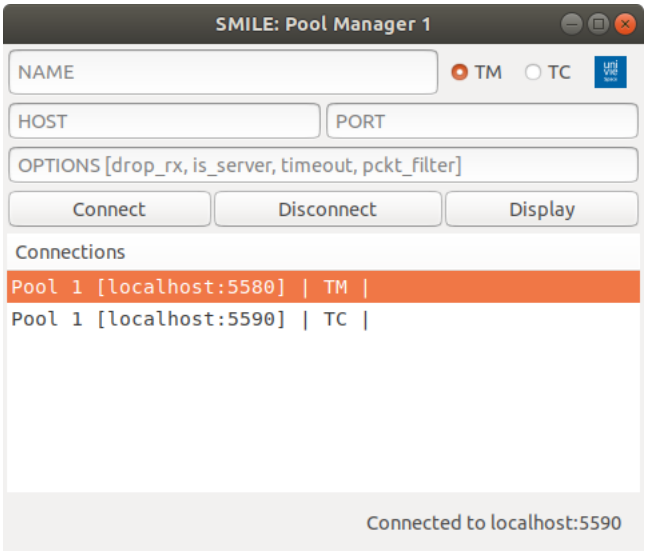

Figure 21: The pool manager GUI, which was created for the SMILE mission (Möslinger, 2020).

It is pretty straightforward to use, since one only has to enter the parameters at the top and press "Connect", to make a connection, or press "Disconnect" and the selected connection will be closed. All opened connections are also listed, which will be updated if a connection is made, where it does not matter if it is made in the GUI or via the terminal. The last feature is the "Display" button, which will show the selected connection in the pool viewer.

The new GUI is, however, not necessary to be started to use the pool manager. It can still run in the background as it was used to be. If needed by another application (mainly pool viewer), it will also be started without a GUI.

Implementation: For the GUI, Gtk is used, just as for the rest of the CCS. For the Connect, Disconnect and Display buttons functions could be used that already existed in the CHEOPS version, but are now easier to use. The window which shows all existing connections is linked to the "connections" list, in which the pool manager keeps track of all available connections and their properties.

Testing: To show that the pool manager GUI is working, it is necessary to start it and just see that it is running. Calling the used functions does not make sense as they are the same as for CHEOPS.

4.1.5 Pool viewer

The pool viewer is very closely linked with the pool manager and in some cases can not complete a given task alone. Therefore, it opens the pool manager in the background (without a GUI) to assist. The user will be notified through a small print out message and an entry into the log file. To give an example, it happens when a previously saved static pool is loaded. It performs a few checks and imports the data to the DB, if it does not already exist there. In the end it returns only the name of the pool to the pool viewer, which then loads the information from the DB and displays it. At first this seems a bit unnecessary to first save it to the DB, only to load it again from there just moments later. However, every incoming pool is first written to the DB to be sure to not lose any information in case of a crash. For a static pool the lost information might not be a problem, since it does not change anymore and is already saved, however, it is easiest to always follow the same procedure and let every application perform the task it was created for.

Furthermore, making a connection out of the pool viewer will now open the pool manager with GUI, since it can perform the same tasks as the window that was opened there in the CHEOPS CCS version.

Simplify the building of User defined Packets *(#15)*

Description: The only other difference in the pool viewer GUI, except the "UNIVIE"- button which was discussed in Section 4.1.1, are the two

buttons in the top left corner, that allow building of user defined packets and parameters. Since all packets are built of parameters those will be explained first. There are two different types, with the only difference being the knowledge of the position. The one type can be plugged into a packet at any wanted position, whereas the others know exactly how many bytes there are between the beginning of the packet and their location. All of them are stored in the ccs_main_config, where they are in two different layers.

User defined Packets are used to decode packets, that have their format not saved in the IDB. Such packets are only needed, if the packet for a specific type has changed or a new packet was created. This is a rarely used functionality, that is only useful in some specific cases. For CHEOPS, it was already possible to set such a packet, but only via the terminal, where it was necessary to pass along a large amount of arguments, which increases with the size of the packet. To have a nice overview, a GUI was created, where one can view all available parameters and select those, that are needed in the packet. Furthermore, also user defined packets can now be included. The two main building differences are, that the packets are either sorted in the order they are given or that the parameters use the locations given in the IDB. However, only user defined parameters with a given position are possible in IDB sorted packets. The CCS allows the user to have complete freedom of how to build the packets, which also means, that the user has to make sure that all parameters are in the desired place they are wanted to be.

At last, while decoding normally, it would always be checked first if a packet can be decoded according to the IDB and if that is not possible it is checked, if a user defined packet exists. If now a in the IDB existing packet type changes and the user would define an alternative, it would still be wrongly decoded according to the IDB. To avoid this and first check for a user defined option, one can toggle the UDEF box in the pool viewer.

Implementation: The first part of implementing the idea was to built a user friendly Gtk GUI, which uses functions that already existed. This was done with the same schema as it was used for all other CCS windows. One thing that did not yet exist was the differentiation between user given and IDB given location of parameters in a newly defined packet. There is now a check implemented that defines which information is necessary to get out of the IDB and how it is saved in the config file. Furthermore, a new Section (in the config file) was created to save the parameters with no given position in a package.

Testing: As for all upgrades that involve the building of a new GUI it is rather difficult to have an appropriate test script to show that it is working,

since the use is supposed to enter the needed information. Therefore, it will be shown that the GUIs are running.

Display pool automatically when started *(#16)*

Description: This is also something that should make the CCS more pleasant to use. The idea is rather simple, as the pool viewer should always check, during starting, if there are any open pools and then show them. It was also discussed, if the pool viewer should always check for new connections while running, but since adding a pool from the pool manager GUI is now very simple, it was decided otherwise. It could also be annoying if just for some testing purpose a connection is made and it always shows up in the viewer, even if there are no packets being sent.

Implementation: It is done by first checking if the pool manager is running and if so, get the information about the pools via D-Bus. It does not matter how many pools are active, all of them will be loaded.

Testing: First it is necessary to have a pool running, which is achieved with a small working script that exists in the CCS folder. It is called "serve_pool_over_sockets.py" and is used to send packets of a previously saved pool via a socket to the CCS. It has to be first started and then the pool manager has to start listening. Then the pool viewer can be started and the pool is shown there. Afterwards it is again closed and a second pool is opened with the same script that was previously used. Then it is again connected to the pool manager and then the pool viewer is restarted. This time both pools will be loaded and just a random one of these two is shown, but it is always possible to switch the view to the other one.

4.1.6 Parameter plotter

Making an independent parameter plotter *(#17)*

Description: The parameter plotter could already be viewed as an individual part for the CHEOPS CCS, but is now a fully individual application and the code can no longer be found in the pool viewer, as an additional file was created where it was transferred to (plotter.py). Some small changes had to be made to get a standalone Gtk application, like adding a Gtk main loop, but those do not affect the functionalities at all. Also all general improvements of the CCS have been added (e.g. D-Bus connection, UNIVIE Button,...). However, in general the parameter plotter stayed the same

and can be used the same way as before, not only graphically, but also the code behind it stayed untouched. One of the only differences is that for the CHEOPS version the plotter could not be started if no pool was shown in the pool viewer. This made sense, since using the plotter without a pool is pointless . However, an independent application should not be so dependent on another one, as it only shows the pool that is currently active in the pool viewer. The plotter should also be available if the viewer is closed. It was not possible to change the pool in the plotter. As a result, if multiple pools were open at the same time and the plotter was opened for one of them, it was necessary to close it, change the shown pool in the viewer and open it again.

Implementation: To make the plotter usable as a standalone application, a small functionality was added that lets the plotter change the shown pool while running. This can be done graphically via the Drop Down menu in the top left corner. For this the plotter checks with the pool manager, every time the Menu is called, which pools are currently open and shows them. If the manager is not running, the viewer is checked. It is therefore no problem to start the plotter without a pool, since it can be changed later.

Testing: One could already see in earlier tests (Section 4.1.1) that the plotter is now standalone, as every general improvement is also implemented in it and was therefore as well part of those tests. However, it is still necessary to show this explicitly. At first the plotter is started to show that it is running, afterwards the pool manager is opened and a pool is loaded. Afterwards the plotter is called to change the active pool to the newly loaded one. One can see in the top left corner that the pool has changed. In the end both applications are closed again and the test is finished.

4.1.7 Parameter monitor

The monitor has a limited, but important purpose of use, where everything worked already perfect for CHEOPS. Therefore, it was not found necessary to change or add anything and concerning functionalities it stayed the same. The only changes are the same that have been made to the whole CCS and are described in Section 4.1.1 (D-Bus connection, UNIVIE Button,...).

4.2 Future Prospect

The University of Vienna has already confirmed to participate in the software development for future astronomic satellite missions (e.g. ARIEL, PLATO,

ATHENA,...). The good news for the future of the CCS is that all of those missions will need some version of it. However, it is not yet clear how much in common those future CCS versions will have with the current one. The main reason for this is that the requirements for the on-board software, and therefore also the CCS, are not yet set in stone. There might still be some changes coming, which makes it nearly impossible to define what upgrades are necessary for the CCS. However, one of the future major updates will be the change from the older PUS to the newly developed PUS-C standard, but it is not clear for which mission this will be the case. Summed up, one could say that the future of the CCS is secured, but some changes will be needed to use it for future missions.

5 Summary

SMILE is an ESA and CAS joint S-class mission that will be launched in 2024. It will increase our knowledge of the interaction between the Earth's magnetic field and the solar wind. With its four different instruments it will derive the first full chain of events that drive the Earth-Sun connection. The satellite is currently in the development phase, where the University of Vienna is participating by developing the Instrument Flight Software. To be able to communicate at every point with the satellite software a tool called CCS was created. This was already done for the previous mission called CHEOPS, but was upgraded to work for SMILE as well. Its main purposes are to store and show TM packets that have been received from the satellite and send TC to the software to check if everything is working fine. It provides simple, but informative ways to show the received data, which is either packet by packet or by listing the timeline of each packet parameter in a diagram.

The CCS consists of six main components:

- *Editor:* (editor.py) Is the main window, CCS can be completely controlled from here, mostly used to write and execute scripts.
- *Pool manager:* (pus_datapool.py) Manages the connections to the satellite software and stores, receives and sends data.
- *Central Function Library (CFL):* (central_function_lib.py) Stores all the main functions that are used by multiple parts and the user. However, doesn't have a GUI.
- *Pool viewer:* (poolview_sql.py) Reads from database and displays the received and sent packages.

- *Parameter monitor:* (monitor.py) Monitors individual parameters and warns the user if something isn't right.
- *Parameter plotter:* (plotter.py) Plots package parameters and shows their temporal course.

All of these components are standalone processes and can be used on their own. However, if multiple ones are running they will communicate with each other via D-Bus. D-Bus is a system Bus that is used on all Linux machines and provides a single channel to which all processes can connect and share data. This not only allows the CCS components to communicate, but the user can always connect to D-Bus and fully control it via a python terminal (as it is implemented in the editor) or script. In the CFL functions have been created that simplifies the usage of D-Bus for the user.

The new CCS version is backwards compatible and can also be used for the already launched mission CHEOPS, for which the original CCS was created. It will also be a part of future missions past SMILE where the University of Vienna will participate such as ATHENA, PLATO and ARIEL.

References

Airbus Website (n.d.). https://www.airbus.com/newsroom/press-releases/de/2019/07/airbus-brings-a-smile-to-esa.html. Accessed: 2019-01-08.

Alchetron Website (2020). https://alchetron.com/CHEOPS. Accessed: 2020-12-08.

Bhardwaj, Anil et al. (2007). "X-rays from solar system objects". eng. In: *Planetary and Space Science* 55.9, pp. 1135–1189. ISSN: 0032-0633.

Boudouridis, A. et al. (2003). "Effect of solar wind pressure pulses on the size and strength of the auroral oval". In: *Journal of Geophysical Research (Space Physics)* 108.A4, 8012, p. 8012. DOI: 10.1029/2002JA009373.

Branduardi-Raymont, G.; C. Wang; L. Dai; E. Donovan; L. Li; S. Sambay (2018). *SMILE Definition Study Report*. ESA. URL: https://www.cosmos.esa.int/documents/1655046/0/SMILE_RedBook_ESA_SCI_2018_1.pdf.

Cantero, Javier (2015). "D-Bus simplification". In: *CC BY-SA 4.0, via Wikimedia Commons*. URL: https://commons.wikimedia.org/wiki/User:JavierCantero.

Cargill, P. J. et al. (2005). "Cluster at the Magnetospheric Cusps". In: *Space Science Reviews* 118.1, pp. 321–366. ISSN: 1572-9672. DOI: 10.1007/s11214-005-3835-0. URL: https://doi.org/10.1007/s11214-005-3835-0.

CAS Website (n.d.). http://www.bcas.cas.cn/perspective/201905/t20190517_209849.html. Accessed: 2019-01-14.

Dungey, J. W. (1961). "Interplanetary Magnetic Field and the Auroral Zones". In: *Phys. Rev. Lett.* 6 (2), pp. 47–48. DOI: 10.1103/PhysRevLett.6.47. URL: https://link.aps.org/doi/10.1103/PhysRevLett.6.47.

ESA Website/ SMILE mission (n.d.). https://sci.esa.int/s/wKe2mK8. Accessed: 2019-01-04.

ESA Website/SCOS-2000 (2020). https://www.esa.int/Enabling_Support/Operations/gse/SCOS-2000. Accessed: 2020-08-08.

Gonzalez, Walter D., Bruce T. Tsurutani, and Alicia L. Clúa de Gonzalez (1999). "Interplanetary origin of geomagnetic storms". In: *ssr* 88, pp. 529–562. DOI: 10.1023/A:1005160129098.

Henderson, M. G. et al. (2006). "Substorms during the 10-11 August 2000 sawtooth event". In: *Journal of Geophysical Research (Space Physics)* 111.A6, A06206, A06206. DOI: 10.1029/2005JA011366.

Henderson, Michael Gerard (1994). "Implications of Viking Imager Results for Substorm Models". PhD thesis. UNIVERSITY OF CALGARY (CANADA).

Hsu, T. and R. L. McPherron (2002). "Average characteristics of triggered and non-triggered substorm-like" events: Can a substorm occur without triggering?" In: *AGU Fall Meeting Abstracts*. Vol. 2002, SM71A–0561.

Huang, Chao-Song (2002). "Evidence of periodic (2-3 hour) near-tail magnetic reconnection and plasmoid formation: Geotail observations". In: *grl* 29.24, 2189, p. 2189. DOI: 10.1029/2002GL016162.

Kalmoni, Nadine et al. (2018). "A direct diagnosis of the plasma waves responsible for the explosive energy release of substorm onset". In: *42nd COSPAR Scientific Assembly*. Vol. 42, pp. D3.6-15-18.

Kerschbaum, Franz (2020). *personal collection*.

Krasnopolsky, Vladimir, Jason Greenwood, and Phillip Stancil (2004). "X-Ray and extreme ultraviolet emissions from comets". eng. In: *Space Science Reviews* 113.3, pp. 271–374. ISSN: 0038-6308.

Mecina, Marko and Roland Ottensamer (2018). "CCS User Manual". In: *CHEOPS-UVIE-UM-002*, Issue 1.6.2.

Milan, S. E., J. S. Gosling, and B. Hubert (2012). "Relationship between interplanetary parameters and the magnetopause reconnection rate quantified from observations of the expanding polar cap". In: *Journal of Geophysical Research (Space Physics)* 117.A3, A03226, A03226. DOI: 10.1029/2011JA017082.

Milan, S. E. et al. (2003). "Variations in the polar cap area during two substorm cycles". In: *Annales Geophysicae* 21.5, pp. 1121–1140. DOI: 10.5194/angeo-21-1121-2003. URL: https://www.ann-geophys.net/21/1121/2003/.

Milan, Stephen E. (2009). "Both solar wind-magnetosphere coupling and ring current intensity control of the size of the auroral oval". In: *grl* 36.18, L18101, p. L18101. DOI: 10.1029/2009GL039997.

Möslinger, Dominik (2020). *personal collection*.

Motoba, Tetsuo et al. (2012). "Magnetic conjugacy of northern and southern auroral beads". In: *grl* 39.8, L08108, p. L08108. DOI: 10.1029/2012GL051599.

Palmroth, M. et al. (2001). "Cusp and magnetopause locations in global MHD simulation". In: *Journal of Geophysical Research: Space Physics* 106.A12, pp. 29435–29450. DOI: 10.1029/2001JA900132. eprint: https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/2001JA900132. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2001JA900132.

Pitout, F. et al. (2006). "Cluster survey of the mid-altitude cusp: 1. size, location, and dynamics". In: *Annales Geophysicae* 24.11, pp. 3011–3026. DOI: 10.5194/angeo-24-3011-2006.

Raab, W et al. (2016). "SMILE: A joint ESA/CAS mission to investigate the interaction between the solar wind and Earth's magnetosphere". eng. In: *In: Proceedings of Space Telescopes and Instrumentation 2016: Ultraviolet to Gamma Ray. Society of Photo-Optical Instrumentation Engineers (SPIE): Edinburgh*. Society of Photo-Optical Instrumentation Engineers (SPIE). URL: http://discovery.ucl.ac.uk/1537401/1/Branduardi%20Raymont_Raab%252B_SPIE_2016_SMILE%28VoR%29.pdf.

Robertson, I. P. and T. E. Cravens (2003). "X-ray emission from the terrestrial magnetosheath". In: *Geophysical Research Letters* 30.8, n/a–n/a. ISSN: 0094-8276.

Robertson, I. P. et al. (2012). "Solar wind charge exchange during geomagnetic storms". eng. In: *Astronomische Nachrichten* 333.4, pp. 309–312. ISSN: 0004-6337.

Taguchi, S. et al. (2009). "HF radar polar patch and its relation with the cusp during B_Y-dominated IMF: Simultaneous observations at two altitudes". In: *Journal of Geophysical Research (Space Physics)* 114.A2, A02311, A02311. DOI: 10.1029/2008JA013624.

Team, ESA SCOS-2000 (2010). "SCOS 2000 User Manual". In: URL: http://emits.sso.esa.int/emits-doc/Annex-J_S3-EGOS-MCS-S2K-ICD-0001_I6.2_SCOS2000-DI-ICD.pdf.

Wallace, Kotska et al. (2005). "Development of micro-pore optics for x-ray applications". In: *Optics for EUV, X-Ray, and Gamma-Ray Astronomy II*. Ed. by Oberto Citterio and Stephen L. O'Dell. Vol. 5900. International Society for Optics and Photonics. SPIE, pp. 309–318. DOI: 10.1117/12.614939. URL: https://sci.esa.int/documents/34490/36224/1567255130848-SPIE-HEO-manuscript-DRAFT.pdf.

A Test Scripts

```
import subprocess   # Start and Stop processes
import time         # Sleep command

# All now individual application files
application_files = ['editor.py', 'poolview_sql.py', '
    pus_datapool.py', 'monitor.py', 'plotter.py']

for app in application_files:
    process = subprocess.Popen(['python3', app]) # Start
    process
    time.sleep(10)  # Wait so it can be seen that it is
    running
    subprocess.Popen.kill(process)   # Kill process
    time.sleep(2)   # Short break before the next application
```

Test 1: All parts of the CCS should be independent Applications

```
import ccs_function_lib as cfl # Central Function Library
import subprocess   # Start and Stop processes
import time         # Sleep command

# Application Files and their names
applications = [['editor', 'editor.py'], ['poolviewer', '
    poolview_sql.py'], ['poolmanager', 'pus_datapool.py'], ['
    monitor', 'monitor.py'], ['plotter', 'plotter.py']]

for app in applications:
    process = subprocess.Popen(['python3', app[1]])  # Start
    process
    time.sleep(2)
    connection = cfl.dbus_connection(app[0]) # Make
    connection
    result = connection.ConnectionCheck() # Call Function
    print(app[0] + ': ' + result) # Print Result
    subprocess.Popen.kill(process) # Kill process
    time.sleep(1)
```

Test 2: All Applications should exchange Data with D-Bus

```
import ccs_function_lib as cfl # Central Function Library
import subprocess   # Start Stop processes
import time         # Sleep command

# Application Files and their names and their variable names
    in the terminal
applications = [['editor', 'editor.py', 'editor'], ['
    poolviewer', 'poolview_sql.py', 'pv'], ['poolmanager', '
```

```
    pus_datapool.py', 'pmgr'], ['monitor', 'monitor.py', '
    monitor'], ['plotter', 'plotter.py', 'paramplot']]
processes = {}

for app in applications:
    process = subprocess.Popen(['python3', app[1]])  # Start
    process
    time.sleep(2)
    processes[app[0]] = process
    editor = cfl.dbus_connection(applications[0][0]) # Make
    connection to the editor
    editor.Functions('_to_console', app[2] + '.Functions("
    get_communication")')
    time.sleep(1)

# Check if it also works if editor is not started first
subprocess.Popen.kill(processes[applications[0][0]]) # Kill
    editor
process = subprocess.Popen(['python3', applications[0][1]])
    # Start editor
time.sleep(10)
processes[applications[0][0]] = process

for app in applications[1:]:
    editor = cfl.dbus_connection(applications[0][0]) # Make
    connection to the editor
    editor.Functions('_to_console', app[2] + '.Functions("
    get_communication")')
    time.sleep(1)
    subprocess.Popen.kill(processes[app[0]])  # Kill process

subprocess.Popen.kill(processes[applications[0][0]]) # Kill
    editor
```

Test 3: Still complete control over CCS in the IPython terminals

```
import ccs_function_lib as cfl # Central Function Library
import subprocess   # Start Stop processes
import time         # Sleep command

# Application Files and their names and their variable names
    in the terminal
applications = [['editor', 'editor.py', 'editor'], ['
    poolviewer', 'poolview_sql.py', 'pv'], ['poolmanager', '
    pus_datapool.py', 'pmgr'], ['monitor', 'monitor.py', '
    monitor'], ['plotter', 'plotter.py', 'paramplot']]

for app in applications:
    run_pv1 = subprocess.Popen(['python3', app[1]])  # Start
    first instance
```

```
    run_pv2 = subprocess.Popen(['python3', app[1]])  # Start
   second instace
    time.sleep(2)
    con1 = cfl.dbus_connection(app[0], 1) # Connect to first
   instance
    con2 = cfl.dbus_connection(app[0], 2) # Connect to second
    instance
    con1.Functions('get_communication') # Call a method,
   first instance
    communication1 = con2.Functions('get_communication') #
   Call a method, second instance and get main communication

    cfl.change_main_communication(app[0], 2, 'SMILE', None) #
    Change main communication

    communication2 = con2.Functions('get_communication') #
   Get Main communication
    #Show the difference
    print('Main instance of ' +app[0]+ ': ', communication1[
   app[0]])
    print('Main instance of ' +app[0]+ ' after change: ',
   communication2[app[0]])

    run_editor = subprocess.Popen(['python3', 'editor.py'])
   # Start editor
    time.sleep(2)
    # Call a method in both pool viewers from the terminal
    editor = cfl.dbus_connection('editor')
    editor.Functions('_to_console', app[2]+'.Functions("
   get_communication")')
    editor.Functions('_to_console', app[2]+'2.Functions("
   get_communication")')
    time.sleep(2)

    # Kill all applications
    subprocess.Popen.kill(run_pv1)
    subprocess.Popen.kill(run_pv2)
    subprocess.Popen.kill(run_editor)
```

Test 4: Multiple Instances of one Application running at the same Time

```
import  subprocess    # Start  Stop  processes
import  time           # Sleep  command

# Part 1 -- CHEOPS
pv = subprocess.Popen(['python3', 'poolview_sql.py', '
   Test_Mantis_2153.tmpool'])   # Start pool viewer
time.sleep(10)
subprocess.Popen.kill(pv)

```

```
# Part 2 -- SMILE
pv = subprocess.Popen(['python3', 'poolview_sql.py', 'sxidata.bin'])   # Start pool viewer
time.sleep(10)
subprocess.Popen.kill(pv)
```

Test 5: Compatibility with CHEOPS

```
import ccs_function_lib as cfl # Central Function Library
import subprocess    # Start Stop processes
import time           # Sleep command

# Application Files and their names and their variable names in the terminal
applications = [['editor', 'editor.py'], ['poolviewer', 'poolview_sql.py'], ['poolmanager', 'pus_datapool.py'], ['monitor', 'monitor.py'], ['plotter', 'plotter.py']]
# Save all process instances
processes = []

#Start every application for both Main Instances
for app in applications:
    proc = subprocess.Popen(['python3', app[1]])  # Start Application, Main Instance 1
    processes.append(proc)  # Save the instance to close it later
    time.sleep(0.5)
    proc = subprocess.Popen(['python3', app[1], '-SMILE_2-'])  # Start Application, Main Instance 2
    processes.append(proc)  # Save the instance to close it later
    time.sleep(2)

for app in applications:
    con1 = cfl.dbus_connection(app[0], 1) # Make Connection
    con2 = cfl.dbus_connection(app[0], 2) # Make Connection

    # Changing communication will not be possible
    con1.Functions('change_communication', app[0], 2) # Change Communication
    result = cfl.Functions(con1, 'get_communication')
    print('The main communicaiton is still to instance ' + str(result[app[0]]) + ' not 2')
    time.sleep(1)
    con2.Functions('change_communication', app[0], 1) # Change Communication
    result = cfl.Functions(con2, 'get_communication')
    print('The main communicaiton is still to instance ' + str(result[app[0]]) + ' not 1')
    time.sleep(1)
```

```
32
33 # Kill all processes
34 for process in processes:
35     subprocess.Popen.kill(process)
```

Test 6: Multiple Instances of whole CCS running at the same Time

```
1  import  ccs_function_lib  as cfl # Central  Function  Library
2  import  subprocess    # Start  Stop  processes
3  import  time          # Sleep  command
4  import os             # Get Log FIle
5
6  # Application  Files  and  their  names  and  their  variable
       names in the  terminal
7  applications = [['editor', 'editor.py'], ['poolviewer', '
     poolview_sql.py'], ['poolmanager', 'pus_datapool.py'], ['
     monitor', 'monitor.py'], ['plotter', 'plotter.py']]
8  # Save all process instances
9  processes = []
10
11 #Start every application for both Main Instances
12 for app in applications:
13     proc = subprocess.Popen(['python3', app[1]])  # Start
     Application, Main Instance 1
14     processes.append(proc)  # Save the instance to close it
     later
15     time.sleep(0.5)
16     proc = subprocess.Popen(['python3', app[1], '-SMILE_2-'])
       # Start Application, Main Instance 2
17     processes.append(proc)  # Save the instance to close it
     later
18     time.sleep(2)
19
20 for app in applications:
21     con1 = cfl.dbus_connection(app[0], 1) # Make Connection
22
23     # Changing communication will cause Log Entry
24     cfl.Functions(con1, 'change_communication', app[0], 2) #
     Change Communication
25     result = cfl.Functions(con1, 'get_communication')
26     time.sleep(1)
27
28 # Kill all processes
29 for process in processes:
30     subprocess.Popen.kill(process)
31
32 # Open an show latest log File entries
33 latest_log = max(os.listdir(str(os.getcwd()) + '/logs'))
34 a_file = open('logs/' + str(latest_log), "r")
35 lines = a_file.readlines()
```

```
last_lines = lines[-5:]
for i in last_lines:
    print(i)
```

Test 7: Centralize the Logging process for all Applications

```
import  ccs_function_lib  as cfl # Central  Function  Library
import  subprocess    # Start  Stop  processes
import  time          # Sleep  command

# Application Files and their names and their variable names in the terminal
applications = [['editor', 'editor.py'], ['poolviewer', 'poolview_sql.py'], ['poolmanager', 'pus_datapool.py'], ['monitor', 'monitor.py'], ['plotter', 'plotter.py']]

for  app in applications:
    process = subprocess.Popen(['python3', app[1]]) # Start process
    time.sleep(2)

    con = cfl.dbus_connection(app[0])
    cfl.Functions(con, 'on_univie_button', True) # Open the Univie Button Menu
    time.sleep(10)  # Wait so it can be seen that it is open

    subprocess.Popen.kill(process)    # Kill process
    time.sleep (0.5)    # Short break before the next application
```

Test 8: The UNIVIE Button

```
import  ccs_function_lib  as cfl # Central  Function  Library
import  subprocess    # Start  Stop  processes
import  time          # Sleep  command

pv_process = subprocess.Popen(['python3', 'poolview_sql.py'])
    # Start pool viewer
pmgr_process = subprocess.Popen(['python3', 'pus_datapool.py'
    ]) # Start pool manager
time.sleep(2)
pv = cfl.dbus_connection('poolviewer')
pv.Functions('load_saved_pool', 'sxidata.bin', 'PUS') # Load PUS pool
time.sleep(10)
pv.Functions('load_saved_pool', 'rx_stream.dat', 'SPW') # Load RMAP/SPW pool
time.sleep(10)

subprocess.Popen.kill(pv_process)
```

```
subprocess.Popen.kill(pmgr_process)
```

Test 9: Implementation of Different Packet Types than PUS

```
import  ccs_function_lib  as cfl # Central  Function  Library

python_values = ['Hello World', 42, True]

dbus_values = cfl.python_to_dbus(python_values)

print(dbus_values)
```

Test 10: Building a Central Function Library without a passed Instance

```
# Two files that should be checked
files = ["ccs_function_lib.py", "pus_datapool.py"]

for file in files:
    struct = 0 # Amount of structs found
    bit = 0 # Amount of bitstrings found
    a_file = open(file, "r") # Open file
    lines = a_file.readlines()
    for line in lines:
        # Ignore all comments
        if line.startswith('#'):
            continue
        if "bitstring" in line:
            bit += 1
        if "struct" in line:
            struct += 1

    print('Bitstring usage: ' + str(bit) + ' times')
    print('Struct usage: ' + str(struct) + ' times')
```

Test 11: Change decoding from Bitstring Module to Struct Module

```
import  ccs_function_lib  as cfl # Central  Function  Library
import  subprocess    # Start  Stop  processes
import  time          # Sleep  command

ed_process = subprocess.Popen(['python3', 'editor.py']) #
    Start editor
time.sleep(2)
ed = cfl.dbus_connection('editor') # Connect to editor

ed.Functions('_to_console', 'while True: \n pass \n') # Send
    Endless loop to terminal
time.sleep(10)
ed.Functions('restart_terminal') # Restart terminal
time.sleep(10)
```

```
subprocess.Popen.kill(ed_process) # Close editor
```

Test 12: Restart only the IPython terminal

```
import  ccs_function_lib  as cfl # Central  Function  Library
import  subprocess    # Start  Stop  processes
import  time          # Sleep  command

ed_process = subprocess.Popen(['python3', 'editor.py']) #
    Start editor
time.sleep(2)
ed = cfl.dbus_connection('editor') # Connect to editor

ed.Functions('_to_console', '"This is the first command"',
    False) # Send Command but do not execute
time.sleep(10)
ed.Functions('_to_console', '"This is the second command"',
    True) # Send Command and execute
time.sleep(10)
subprocess.Popen.kill(ed_process) # Close editor
```

Test 13: Communication with the IPython terminal

```
import  subprocess    # Start Stop processes
import  time          # Sleep command

ed_process = subprocess.Popen(['python3', 'pus_datapool.py'])
    # Start pool manager
time.sleep(10)
subprocess.Popen.kill(ed_process)
```

Test 14: pool manager GUI

```
import  ccs_function_lib  as cfl # Central Function Library
import  subprocess    # Start Stop processes
import  time          # Sleep command

# Show Parameter GUI
pv_process = subprocess.Popen(['python3', 'poolview_sql.py'])
    # Start pool viewer
time.sleep(2)
pv = cfl.dbus_connection('poolviewer')
pv.Functions('add_decode_parameter', True, ignore_reply=True)
    # Open Parameter GUI
time.sleep(10)
subprocess.Popen.kill(pv_process) # Close pool viewer

# Show Packets GUI
pv_process = subprocess.Popen(['python3', 'poolview_sql.py'])
    # Start pool viewer
```

```
time.sleep(2)
pv = cfl.dbus_connection('poolviewer')
pv.Functions('add_new_user_package', True, ignore_reply=True)
    # Open Packets GUI
time.sleep(10)
subprocess.Popen.kill(pv_process) # Close pool viewer
```

Test 15: Simplify the building of User defined Packets

```
import  ccs_function_lib  as cfl # Central Function Library
import  subprocess    # Start Stop processes
import  time          # Sleep command

pmgr_process = subprocess.Popen(['python3', 'pus_datapool.py'
    ]) # Start pool manager
time.sleep(2)
pmgr = cfl.dbus_connection('poolmanager') # Connect to pool
    manager

pool1_process = subprocess.Popen(['python3', '
    serve_pool_over_sockets.py', 'Test_Mantis_2153.tmpool','
    5570']) # Start pool manager
time.sleep(2)

pmgr.Functions('connect', 'Test_Pool_1', 'localhost', 5570)

pv_process = subprocess.Popen(['python3', 'poolview_sql.py'])
    # Start pool viewer
time.sleep(10)
subprocess.Popen.kill(pv_process)

pool2_process = subprocess.Popen(['python3', '
    serve_pool_over_sockets.py', 'Test_Mantis_2153.tmpool', '
    5580']) # Start pool manager
time.sleep(2)

pmgr.Functions('connect', 'Test_Pool_2', 'localhost', 5580)

pv_process = subprocess.Popen(['python3', 'poolview_sql.py'])
    # Start pool viewer
time.sleep(10)
subprocess.Popen.kill(pv_process)
subprocess.Popen.kill(pmgr_process)
subprocess.Popen.kill(pool1_process)
subprocess.Popen.kill(pool2_process)
```

Test 16: Display pool automatically when started

```

import ccs_function_lib as cfl # Central Function Library
```

```
import subprocess    # Start Stop processes
import time          # Sleep command

app = [['plotter.py', 'plotter'], ['poolview_sql.py', '
    poolviewer'], ['pus_datapool.py', 'poolmanager']]

plot_proc = subprocess.Popen(['python3', app[0][0]])  # Start
     the Plotter
pv_proc = subprocess.Popen(['python3', app[1][0]])  # Start
    the Viewer
pmgr_proc = subprocess.Popen(['python3', app[2][0]])  # Start
     the Manager
time.sleep(4)
plot = cfl.dbus_connection(app[0][1]) # Connect to Plotter
pv = cfl.dbus_connection(app[1][1])  # Connect to Viewer
pmgr = cfl.dbus_connection(app[2][1])  # Connect to Manager
time.sleep(2)
cfl.Functions(pv, 'load_pool', None, 'Test_Mantis_2153.tmpool
    ')
time.sleep(2)
cfl.Functions(plot, 'pool_changed', None, 'Test_Mantis_2153.
    tmpool', ignore_reply=True)
time.sleep(2

subprocess.Popen.kill(pv_proc)
subprocess.Popen.kill(plot_proc)
subprocess.Popen.kill(pmgr_proc)
```

Test 17: Making an independent parameter plotter

A Abstract

Das Central Checkout System (CCS), wird während der Test- und Entwicklungsphase einer Satellitensoftware benötigt. Zu seinen Hauptaufgaben gehören das Empfangen, Speichern und Anzeigen von TM Packeten, die von der Satellitensoftware versendet werden, sowie das Senden und Erstellen von TC Packeten, um Befehle des Benutzers an die Software weiterzuleiten. Es ermöglicht dem Benutzer mit Hilfe einer graphischen Benutzeroberfläche einfach mit der Satellitensoftware zu kommunizieren. Jedoch hat eine jede Mission andere Anforderungen. Deshalb muss immer eine eigene Software und damit auch ein eigenes CCS entwickelt werden. In dieser Arbeit wird die Weiterentwicklung des CCS, von der bereits gestarteten Mission CHEOPS, auf die von ESA und CAS gemeinsam entwickelte Satellitenmission SMILE, beschrieben.

The Central Checkout System (CCS), is used during the development and testing phase of a satellite software. It's main tasks are to receive, save and show TM packets, which contain data from the satellite software, as well as build and send TC packets, which are commands to the software. It makes the communication, for the user, a lot easier due to its own GUI. However, every mission has it's own specification, which necessitates the development of a individual satellite software and CCS. In this work the further development of the CCS, from the CHEOPS mission, for the ESA and CAS joined mission SMILE, is described.