1 Uplink and Downlink (non-DP)

1.1 Ground Segment Interfaces

The HSC used leased lines to provide a reliable connection to the MOC and the ICCs. These were later replaced by use of the Internet. Leased lines may no longer be appropriate in many cases because the reliability and performance of the Internet is now so good. If leased lines *are* used, it is important to be able to switch rapidly between them and the Internet, so there is redundancy (Lesson H14).

The work that went into defining and reviewing the MOC-SOC ICDs generally resulted in stable interfaces with very few problems (LessonH19). The overall telecommand volume was not specified in the ICDs, which led to the need for changes to the on-board CDMS software and MOC MCS software (Lesson H131).

It is difficult to avoid problems with leap seconds because the ground segment as a whole contains many legacy systems that do not handle them properly. There are some precautions that can be taken to minimize problems (Lesson H18 & H19).

For the ICC-MOC interface, it was invaluable having ICC systems at the MOC (Lesson H137).

The Planning Skeleton Files (PSFs) for mission planning used the concept of constraints, taken from Integral (Lesson H25). This approach is more flexible than rigid windows and allows more efficient scheduling. This approach could be developed further for future missions.

Globally-unique identifiers were needed for numbering observations, telecommands, etc. With hindsight, it would have been better to use the Operational Day (OD) number as a prefix. This would have simplified handling of some contingency situations (Lesson H23).

On-Board Time (OBT) was nominally TAI (+ drift). When this was changed close to launch it caused some confusion. It would have been better to treat OBT simply as a counter, represented by a separate data-type (Lesson H20).

1.2 Instrument Commanding (CUS)

The Common Uplink System (CUS) provided a powerful instrument commanding system. It was a key component of the Herschel “smooth transition” concept and supported both routine observing modes and “one-off” test modes (Lesson H30).

Experience with the CUS has identified a number of areas that could be improved for future missions. In particular, a better versioning mechanism is needed (Lesson H36 & H106). More Eclipse-like features in the GUI would help script development and testing (Lesson H47). There are also some features that could usefully be added to the scripting language. Test harnesses for scripts would also be useful (Lesson H202).

HIFI used the CUS in a way that that had not originally been intended, leading to a number of problems (Lessons H146 & H62). The HIFI sequencer optimised the parameters of an observation by repeatedly executing a CUS script within an optimisation loop. Problems arose from trying to use the CUS language as a programming language when it is was designed as an instrument commanding language. It also required a high-performance server configuration to support AO calls. If a sequencer-like feature is needed for a future mission, it needs to be designed-in.

1.3 Mission Planning Software (SMPS/HILTS)

The Herschel Scientific Mission Planning System (SMPS) did not support planning of contingency scenarios or parallel working by multiple mission planners. It is nice to support these but not straightforward because of interdependencies between ODs and the need to prevent an observation being scheduled more than once. The approach we used was a staging system that provided a “sandbox” for experimentation, with the ability to export schedules to the operational system. This was not perfect, but worked quite well in practice (Lesson H29).

The SMPS was designed to handle the day-by-day scheduling. We then added a long-term planning system (HILTS) to visualise and optimise planning over longer periods. Ideally, more of the long-term planning facilities should have been available in the SMPS (Lesson H60).

There were some aspects of the on-board Attitude Control and Monitoring System (ACMS) behaviour that were not completely predictable (e.g. slew type). This led to problems because the ground-based mission planning software needs to know exactly what will happen in advance. ACMS behaviour should be fully predictable (Lesson H5).

We implemented fast and accurate constraint checking without the need for “sky bins” as used on some previous missions. This maximised the number of observations available for scheduling. Constraints were pre-computed where possible, leading to a very fast implementation (Lesson H24).

We used CCSDS standards for orbit files, etc. The use of international standards helps interchange of data with other organisations (e.g. JPL/Horizons) and made our software more reusable (Lesson H26).

1.4 Telemetry Ingestion and Data Distribution

We had problems throughout the whole mission determining when we had received all the telemetry for a given observation. Although this seems simple, there were many subtle factors that complicated checking for missing packets (Lesson H13).

Data ingestion and distribution ran unattended and status monitors alerted the computer operator when there was a problem. Automatically generated emails and web pages were used to provide visibility of the system status. These details need to be considered as part of the requirements of a system. In our case, the requirements did not go far enough (Lessons H16 & H90).

1.5 Operations Database

The original concept was that we would use an object database to hold all uplink and downlink data. It soon became apparent that it was better to use the ESAC archive infrastructure for the processed product data and restrict the object database to uplink and raw data. This has the added advantage that the data remains available after the end of the mission (Lesson H31).

The object database worked reasonably well and simplified some aspects of uplink development. However, it was expensive and it was difficult to find the necessary expertise. The advantages of the object database were not clear-cut. Nowadays, it is possible to use object persistence with a relational database (e.g. using JPA) (Lesson H38).

We implemented a database replication mechanism to propagate data to the ICCs. This turned out to have many other uses, such as keeping standby, test and staging systems synchronised (Lesson H158).

Some data in the object database needed to be versioned. Versioning is important but care is needed not version objects unnecessarily as it adds a lot of complexity and bloats the database with unused objects (Lesson H33).

It is recommended to build a scalable, reliable server architecture from a number of low-cost machines. We initially bought a high-reliability, high-performance Solaris database server. This was replaced even before launch with a number of low-cost Linux machines (Lesson H17).

Applications that rely on connection to a database can be difficult to develop and test. They are also a problem for working off-line, giving demonstrations at meetings, etc. It is suggested that, where possible, an application should work “out of the box” using an embedded server. It can be configured for operational use simply by setting a configuration property to point at a database server (Lesson H32).

1.6 Miscellaneous

Java was chosen as the programming language for the HCSS. From the uplink perspective, this was a good choice. The extensive class libraries for graphics, networking, etc, helped a lot. It avoided the problems of mixed-language integration and portability that we had experienced on ISO and XMM. The portability was particularly important for proposal submission (HSPOT) (Lesson H12).

Software should be constructed from reusable components. Components that minimise dependencies on one another and make the minimum of assumptions about the mission are potentially reusable and are easier to maintain. This saves work in the long run by making the components more robust, easier to maintain and adapt to changing requirements (Lesson H1).

Software should be designed with the flexibility to cope with contingencies. It must be possible to cope with non-nominal situations. A highly-integrated system built around a database has a danger of being a “black box” that can only handle predefined use-cases. An open system architecture makes it easier to “get in” and work around problems (Lesson H22).