FROM NAPKIN SKETCHES TO FLIGHT SOFTWARE

Johan Hardy[(1)], Arnaud Bourdoux[(2)], Dominique Torette[(3)], Paul Parisis[(4)], Gonzalo Campos Garrido[(5)]

[(1)] *Spacebel, Liège Science Park, Rue des Chasseurs Ardennais 6, B-4031 Angleur, Belgium, johan.hardy@spacebel.be*
[(2)] *Spacebel, Liège Science Park, Rue des Chasseurs Ardennais 6, B-4031 Angleur, Belgium, arnaud.bourdoux@spacebel.be*
[(3)] *Spacebel, Liège Science Park, Rue des Chasseurs Ardennais 6, B-4031 Angleur, Belgium, dominique.torette@spacebel.be*
[(4)] *Spacebel, Liège Science Park, Rue des Chasseurs Ardennais 6, B-4031 Angleur, Belgium, paul.parisis@spacebel.be*
[(5)] *CNES, 18 avenue Edouard Belin, 31401 Toulouse cedex 4, France, gonzalo.camposgarrido@cnes.fr*

Abstract

This paper presents the success story of flight software designed and modelled under Eclipse Papyrus. The presented use case is the on-board software of the main payload of the French satellite TARANIS. The paper describes the model-based design approach followed for the development of the on-board software TARANIS through the following subjects: specification and design of on-board software under Papyrus, UML extensions for embedded C software and code generation from Papyrus models. Finally, this paper presents the achievements of the development and it appeals at further perspectives for Eclipse, Papyrus and next space missions.

Keywords: *Eclipse, Modelling, UML, Papyrus, Space, Embedded Software, Spacecraft, Taranis.*

I. INTRODUCTION

Software architects usually start designing by drawing on paper little sketches or rough diagrams representing the beginnings of the functionalities or the preliminary architecture of a complex software. The sketches and the drawings are then refined with the help of modelling tools. At the end, the code and the documentation are produced with appropriate tools. This is commonly called the Model Based Design (MBD) approach. In the field of software engineering, this is the usual approach followed for complex and large-scale software.

There are a number of existing commercial modelling tools which provide end-to-end features from the specification to the source code. Unlike open source solutions, they often look not flexible enough to cope with all specific user needs. Indeed, it often happens that the modelling tool is not exactly able to provide the needed customisation or even no customisations at all. The mastering of the generation of code may be also an issue with respect to the coding rules, the documentation in the code, etc. Some modelling tools do not provide any code generation at all.

In fact, a lot of circumstances may influence the selection of design strategies for development process and on-board software (OBSW) architecture. In the frame of conventional software standards for space [1], the quality of the source code and the definition of the various models are a daily concern.

Established in 1988, Spacebel successfully designed and developed lots of flight software for different kind of space missions. Spacebel's developers and architects usually use modelling techniques to produce the architecture and the design. These methods were uniquely based on UML and commercial toolsets.

Recently, R&D efforts around Eclipse technologies have encouraged Spacebel to throw the switch of open source solutions. The use of Eclipse solutions and Papyrus [2] to model critical and complex software was a real challenge; here is the success story of TARANIS.

II. THE CONTEXT

TARANIS is a CNES microsatellite that will provide a set of unprecedented measurements of the recently discovered discharges that occur above thunderstorms. These consist mainly in red sprites, blue jets, elves, sprites halos and gigantic jets, named Transient Luminous Events (TLEs) and, on the other hand, the so called Terrestrial Gamma ray Flashes (TGFs). All these phenomena demonstrate the existence of energy transfers between the Earth atmosphere and the space environment.

To accomplish its mission, TARANIS spacecraft will be launched in 2018 from Kourou into a low Earth orbit (~700 km altitude). The minimum mission duration is two years, with an objective of four years. The satellite itself is formed by a CNES Myriade platform plus a dedicated payload. The total mass of the microsatellite is 180 kg and its typical power consumption will reach 80 W. Communication will be established through an S-band flow for telecommands and housekeeping telemetry, and an X-band flow to transmit the scientific TM stored in the on-board mass memory.

Figure 1: TARANIS microsatellite © CNES/Oliver Sattler 2012

TARANIS scientific payload consists in the following eight instruments: micro-cameras, photometers, X-ray and γ-ray detectors, two separated electron detectors, low frequency electrical field detector, high frequency electrical field detector and a magnetic field detector. All these instruments are powered, controlled and interfaced to the platform through a single equipment, the Multi EXperiment Interface Controller (MEXIC). This equipment consists in eight analyser boards to interface the mentioned detectors, a power unit and the MEXIC interface unit (MIU), which is the central element of MEXIC as it manages all scientific instruments. In the frame of an industrial contract with CNES, Spacebel is in charge of MIU Flight Software development and future maintenance.

The main functionalities of MIU Flight Software are: (i) incoming telecommand management, (ii) scientific telemetry conversion, dating and forwarding towards mass memory, (iii) housekeeping telemetry generation and forwarding towards on-board computer, (iv) functional modes management, (v) synchronization, (vi) failure detection, isolation and recovery (FDIR) and (vii) interfaces management. From an operational point of view, these functionalities have to be implemented in such a way that the scientific payload is seen as one instrument (thus synchronizing and getting simultaneous data from all the on-board instruments) but giving the flexibility to operate each instrument independently if needed. In practice, a number of operational modes and sub-modes (i.e. INITBOOT, SERVICING, STANDBY, iSCIENCE, etc.) allow to fulfil the above mentioned functionalities. These modes are spread across the two different software elements, the boot software and the application software, that jointly form the MIU Flight Software.

To correctly develop such kind of complex and critical software product under European standards, Model Based Design is strongly recommended [3].

III. THE CHALLENGE

Switching to an open source solution and a new modeller to produce a complex models was not straight-forward. Obviously, it generated unexpected issues but also changed the mind-set for designing models in general.

Indeed, open source is sometimes seen controversial for production and top management. Managers, quality assurance and production must be convinced that the new solution must do at least the same job with the same features as the old solution but with benefits. Otherwise, it is not worth changing the existing production tools. Changing the tool may change the development process and the latter must be taken with care. In fact, changing the production process can introduce a risk. The argument could be that the risk is minored if the complete processes are mastered from the specification to the code. However, bugs may happen in an open source library, which makes very hard to master the whole solution. Indeed, depending on the reactivity of the open source community, using open source solutions may also introduce a new risk at this level.

In order to satisfy users and lower the risks, the open source community must be an active and rich ecosystem. It needs a large community that distributes and widely uses the solutions (not only in aerospace domain). Furthermore, the community must return feedbacks to improve the solutions. Without the latter requirements and a critical mass of users, it is hard to convince anybody that an open source solution is worth to be adopted.

Fortunately, the Eclipse community follows these open source principles and the community succeeds: Eclipse has more than 250 different open source projects where a lot of working groups (aerospace, defence and security, energy, health care, telecommunications, transportation, etc.) allow for organizations to collaborate in the development of new innovations and solutions. In particular, Eclipse Papyrus is in continuous improvement.

Beyond the above uncertainties, TARANIS software development faced other issues. The software of TARANIS payload is targeted to be loaded in the IP-Core 8051. This IP-Core is based on the implementation of the 8051 chip developed by Intel in the 80's. This 8-bit processor embedding only 256 bytes of RAM is able to operate on bits. Bits are in fact specified by absolute addresses (no indexed addressing). This memory mapping is probably one of the reasons for the 8051's popularity. The 8051 has been used by CNES in the space industry for years. In fact, the chip had been qualified and hardened for space. At the end of production of the chip, the IP-Core 8051 replaced the chip. Nowadays, some French space missions still use the 8051 as IP-Core on FPGA for spacecraft avionics or specific payloads like TARANIS payload.

The software of TARANIS payload is written in C and compiled with a specific cross compiler for the IP-Core 8051. However, due to a particular instruction set and memory mapping, the C code is not strictly ANSI. Indeed, the language used to develop the software is the C language plus some extensions. Here is, for instance, a declaration of an interrupt routine:

```
void IT_handle_interrupt(void) interrupt VECTOR_ID;
```

The function `IT0_handle_interrupt` is extended by the identifier of the interrupt vector.

Although common C data types (*int*, *short*, *float*, *void*, etc.) can be used with the 8051, they can be qualified as internal data:

```
/* unsigned integer coded on 8bits idata */
typedef unsigned char idata uint8_idata;
```

Or be used as external data:

```
/* Write a byte in external SRAM */
uint8 xdata *data location;
*location = (uint8) byte;
```

Indeed, external memories can be used and interfaced with the processor to extend the poor capacity of the internal RAM (256 bytes). So, this adds another extension with another representation of a data type in C. Furthermore, exotic types like *bit*, *sbit* or *sfr* are again available at runtime.

These particularities are not common and sometimes not expected at design level. But they are important for the final source code. Therefore, the modeller shall be flexible enough to model new extensions for the data types and operations. In addition, the modeller shall provide all the commonalities of the C language: *typedef*, *#define*, *#include*, etc. And, the code generation shall take into account the new extensions with the adequate syntax.

Another concern is the specification of the software. The specification of the functionalities is written while designing the overall architecture of the software. The modeller shall then be able to provide a way to manage requirements and a way to constrain model elements.

One important aspect in aerospace industry is the documentation. A lot of documents must be written and delivered for the project reviews (along the development) and the final acceptance of the software for flight. Detailed documentation of the code, the design and the specification must be produced. The diagrams and the descriptions of the architecture are key to pass quality and critical design reviews. The modeller and the generation of the documentation are thus essential for the success of the project.

Finally, the quality of the generated code, the documentation and the software itself must respond to the needs of the mission, the standards of quality and the criticality of the software. The solution shall also provide features for day-to-day work like team collaboration, comparison of model revisions and merge capabilities.

IV. THE SOLUTION

After long internal discussions at Spacebel and several prototypes of small software, Spacebel selected Eclipse Papyrus as modelling tool because of its customisation features and strong compliance to the UML standard.

For the TARANIS project, since the target language of the software is C plus some 8051 extensions, the semantic of UML did not permit to map the entire semantic in UML. Therefore, two UML profiles and a couple of stereotypes have been created aside of the project. The first profile is called *Embedded C* profile. This profile maps missing UML semantic for the ANSI C language.

The *Embedded C* profile provides the primitive types of the C language:

- signed / unsigned char
- signed / unsigned int
- signed / unsigned short
- signed / unsigned long
- singed / unsigned long long
- float
- double
- void

The profile proposes several stereotypes that extend the UML semantic. Among these stereotypes, we have, for example, the *Pointer* stereotype:

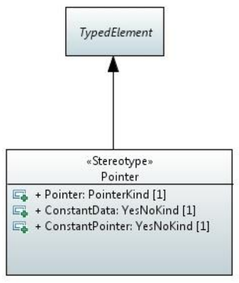

TypedElement
«Stereotype»
Pointer
+ Pointer: PointerKind [1]
+ ConstantData: YesNoKind [1]
+ ConstantPointer: YesNoKind [1]

Figure 2: Stereotype Pointer

The *Pointer* stereotype extends the abstract meta-class *TypedElement* and it provides the extension *Pointer* typed as *PointerKind* (enumeration composed of the literals none, simple and double), *ConstantData* and *ConstantPointer* both typed as *YesNoKind* to specify whether the pointer and/or the data are `const` values.

An interesting extension for the specification of TARANIS software is the stereotype *Requirement*. In UML, the meta-class *Constraint* already provides a property that constrains any model elements. The same meta-class provides also a property to write the sentence of the requirement. Almost everything exists in UML – SysML profile proposes the element *Requirement* but it does not comply exactly with Spacebel's design process – except the traceability with user requirements and the method of verification of the requirement. Thus, extending the meta-class *Constraint* with new properties like *Parent* enables the traceability between user requirements (i.e. CNES requirement baseline) and technical specification (i.e. Spacebel technical requirements). The same stereotype also provides the property *Verification* which specifies how the *Requirement* will be verified (Review, Inspection, Test, Analysis). In doing so, the architectural elements can easily be traced with their requirements and the detailed design can be deduced from the specification:

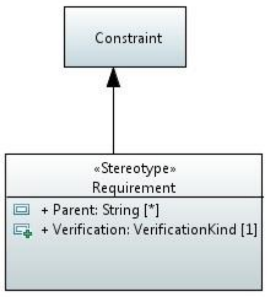


Figure 3: Stereotype Requirement

Other stereotypes like *Array*, *Inline*, *Typedef* … complete the *Embedded C* profile; refer to [4] for complete details.

In addition, a second profile called *core 8051* is used for the particularities of the core 8051 target. The *core 8051* profile provides its primitive types: *bit*, *sbit* and *sfr*. The profile also proposes stereotypes that complement *Embedded C profile* for the core 8051. Among these stereotypes, we have, for example, the *Interrupt* stereotype:

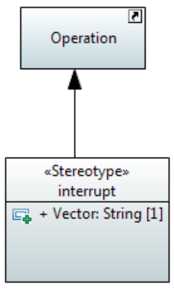
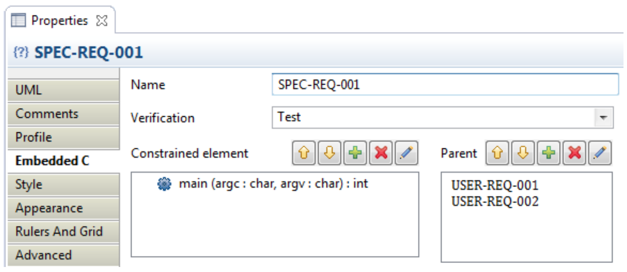


Figure 4: Stereotype Interrupt

The *Interrupt* stereotype extends the meta-class *Operation* to specify whether the *Operation* is a core 8051 interrupt routine. The *Vector* can be specified as property of the stereotype. Other stereotypes like *idata*, *xdata*, *reentrant* … complete the *core 8051* profile.

Thanks to the UML standard, UML profiles and Papyrus, Spacebel's architects achieved extending UML and enriching its semantic with new model elements. Moreover, thanks to Papyrus infrastructure and its possible extensions, the profiles have been enclosed in plugins and fully integrated into the development environment. Indeed, the property view of Papyrus has been customised for the profiles to enhance edition of stereotyped elements:



SPEC-REQ-001

Name: SPEC-REQ-001
Verification: Test
Constrained element: main (argc : char, argv : char) : int
Parent: USER-REQ-001, USER-REQ-002

Figure 5: Customisation of the Property view for a Requirement

The new child menu of Papyrus has also been customised for the profiles to ease the creation of new stereotyped elements:

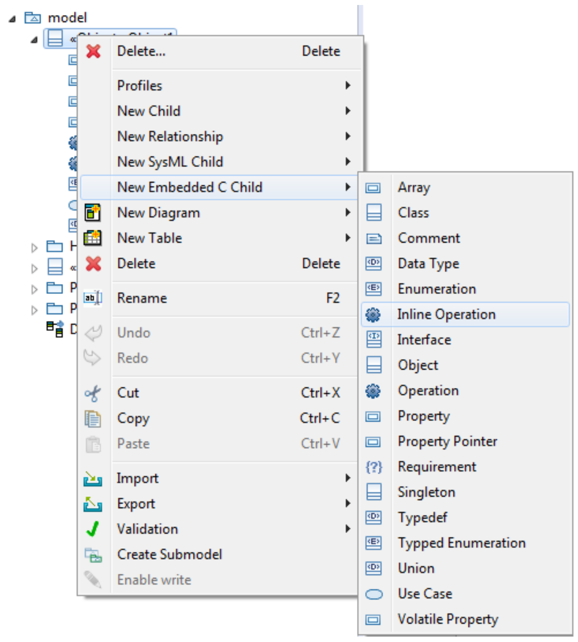


Figure 6: Customisation of the New Child menu

Besides customisations, managing updates and modifications of profiles is also important. The definition of profiles may evolve along the development of the model. Thanks to Papyrus, each version of a profile is conserved with a revision and a date which helps generating new revisions of the profile. Papyrus is also able to manage the revision of a profile for an existing model using the revised profile. The migration of the model to the new revision is automated.

Papyrus includes many diagrams such as activity diagram, communication diagram, component diagram, deployment diagram, package diagram, etc. In fact, all UML diagrams are available. Diagrams and palettes can be customised too according to profile specification.

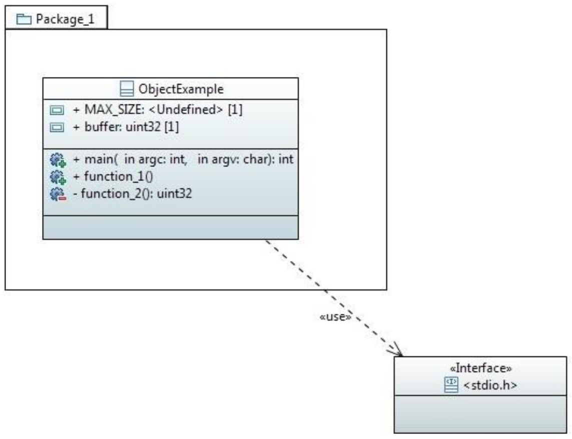


Figure 7: Class diagram example

For example, often used types of diagrams are class diagrams, which are typically diagrams used to describe the overall architecture of software, or state diagrams, which are used to define, for instance, the system modes and their transitions:

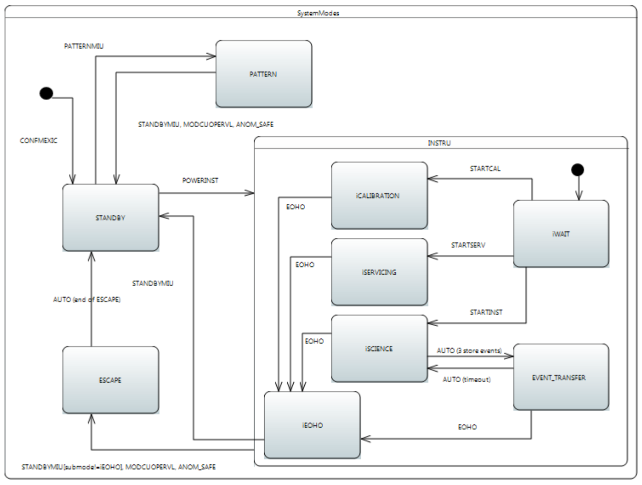


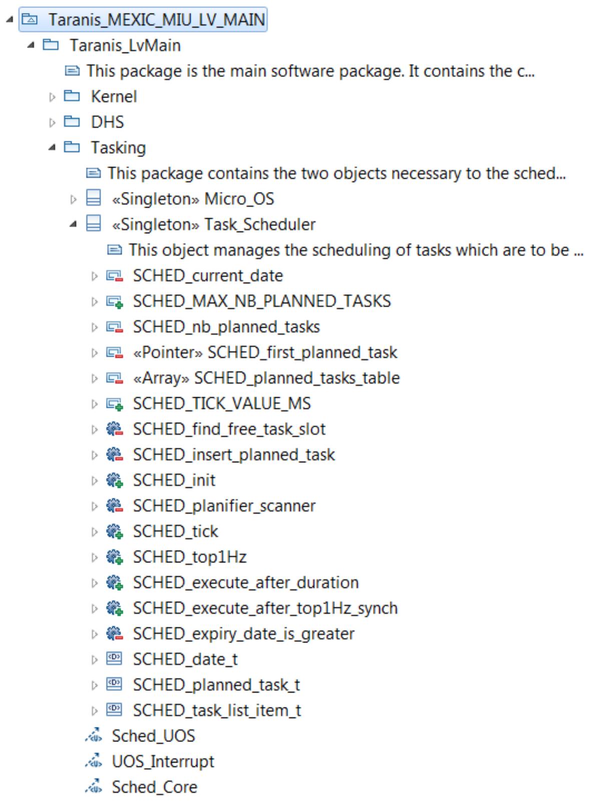
Figure 8: System modes (payload)

Last but not least, at the end of the specification and the design, the code and the documentation are auto-generated from UML. Spacebel uses the techniques of model to text (M2T) transformations based on Eclipse Acceleo for the code and Eclipse Gendoc for the documentation.

Thanks to Acceleo, a reliable *Embedded C* code generator [5] has been developed by Spacebel offering the following features:

- repeatable and reliable generation of code
- preservation of the implementation bodies
- highly documented detailed design (Doxygen)
- compliance to common guidelines of C language in aerospace domain
- traceability of the specification and detailed design
- customisation and configuration of the code generation
- multiple configurations for code generation

From the UML description and the *Embedded C* code generator, all the code (except the body of the functions) is auto-generated with the right syntax and a complete documentation.



```
Taranis_MEXIC_MIU_LV_MAIN
  Taranis_LvMain
    This package is the main software package. It contains the c...
    Kernel
    DHS
    Tasking
      This package contains the two objects necessary to the sched...
      «Singleton» Micro_OS
      «Singleton» Task_Scheduler
        This object manages the scheduling of tasks which are to be ...
        SCHED_current_date
        SCHED_MAX_NB_PLANNED_TASKS
        SCHED_nb_planned_tasks
        «Pointer» SCHED_first_planned_task
        «Array» SCHED_planned_tasks_table
        SCHED_TICK_VALUE_MS
        SCHED_find_free_task_slot
        SCHED_insert_planned_task
        SCHED_init
        SCHED_planifier_scanner
        SCHED_tick
        SCHED_top1Hz
        SCHED_execute_after_duration
        SCHED_execute_after_top1Hz_synch
        SCHED_expiry_date_is_greater
        SCHED_date_t
        SCHED_planned_task_t
        SCHED_task_list_item_t
      Sched_UOS
      UOS_Interrupt
      Sched_Core
```

Figure 9: TARANIS Task Scheduler model tree

All the members are auto-generated in the header or/and source files. The attributes of the class like constants or variables are generated:

```
/*!
 * \brief Maximum number of planned tasks in the task scheduler
 */
#define SCHED_MAX_NB_PLANNED_TASKS 32U

/*!
 * \brief Time in milliseconds between each scheduler tick
 */
#define SCHED_TICK_VALUE_MS 50U

/*!
 * \var SCHED_current_date
 * \brief Current date as seen by the task scheduler.
 *        The seconds field is equal to the number of
 *        Top1Hz received since startup.
 *        The milliseconds field is equal to the number of
 *        milliseconds since last Top1Hz.
 */
static SCHED_date_t SCHED_current_date;
```

```
/*!
 * \var SCHED_nb_planned_tasks
 * \brief Current number of planned tasks.
 */
static uint8 SCHED_nb_planned_tasks;
```

The data types and their descriptions are auto-generated in the header file:

```
/*!
 * \brief Type describing an on-board date for the task scheduler.
 */
typedef struct SCHED_date_t
{
    uint16 seconds; /*! Number of seconds since start */
    uint16 millisec; /*! Number of milliseconds since start */
} SCHED_date_t;

/*!
 * \brief This structure represents a planned task
 */
typedef struct SCHED_planned_task_t
{
    MICROS_task_id_t task_id; /*! Identifier of the task */
    SCHED_date_t expiry_date; /*! Expiry date expressed in the
                                  scheduler time base */
} SCHED_planned_task_t;
```

Prototypes of functions including the documentation are auto-generated as well:

```
/********************* FUNCTION ********************************/
/*!
 * \brief This function schedules the selected task for execution
 *        after the given number of milliseconds.
 *        The expiry date is converted to the (seconds,milliseconds)
 *        scheduler time reference before insertion.
 *
 * \param task_id [in]
 *        The id of the task to be scheduled.
 *
 * \param expiry_date [in]
 *        Time in milliseonds from the current time before execution
 *        of the task.
 *
 * \return return_code_t : The error code.
 *
 * <b>Function detailed processing extracted from source code</b>
 *
 ***************************************************************/
return_code_t SCHED_execute_after_duration (MICROS_task_id_t
task_id, uint16 expiry_date);

/********************* FUNCTION ********************************/
/*!
 * \brief This function initialises the task scheduler.
 *        The current time is set to (0,0), and
 *        the task table is initialised empty:
 *         - All task ids are set to 0 (null task),
 *         - The first task pointer is initialised to NULL,
 *         - The number of planned tasks is set to 0.
 *
 * <b>Function detailed processing extracted from source code</b>
 *
 ***************************************************************/
void SCHED_init (void);
```

And finally the body of functions is written by the developer between protected fields in the source file. Code written this way is preserved if the code is regenerated from the model.

```
/*## operation SCHED_init(void) */
void SCHED_init (void)
{
    /* Start of user code for SCHED_init */
    uint8 i;

    /* Init all variables of SCHED module */
    SCHED_current_date.seconds = 0;
    SCHED_current_date.milliseconds = 0;
    SCHED_nb_planned_tasks = 0;
    SCHED_first_planned_task = NULL;

    for (i = 0; i < SCHED_MAX_NB_PLANNED_TASKS; i++)
    {
      SCHED_planned_tasks_table [i].task.task_id =
                    M_TID_RESERVED_NULL;
      SCHED_planned_tasks_table [i].task.expiry_date.seconds = 0;
      SCHED_planned_tasks_table [i].task.expiry_date.millisec = 0;
      SCHED_planned_tasks_table [i].next = NULL;
    }
    /* Register timer0 callback */
    CORE_timer0_register_callback (SCHED_tick);
    /* End of user code for SCHED_init */
}
```

In a similar fashion, the Software Requirement Document (SRD) and the Architectural Design Document (ADD) including diagrams and all related traceability matrixes have been successfully auto-generated as well by using Gendoc (in a Microsoft Word format ready to deliver).

V. THE BENEFITS

The development of the TARANIS MIU software has been a pilot project at Spacebel for deploying the open source and Eclipse tool chain. The reasons for selecting this project, and the benefits that the toolchain brought to the product, are summarized here.

First of all, the size of the project made it suitable. With a few thousands of lines of code, it is neither too small, which might have reduced the impact of the achievement, nor too large, which could have been too much of a risk for the company.

Second, the contract with CNES mentions the possibility of the complete takeover of the software responsibility after the production phase, into the maintenance phase. This requires the software development environment to be a deliverable in addition to the produced software. In this scheme, the open source environment has both the advantages of not requiring the procurement of possibly expensive licences, and of having an active community which can possibly contribute with improvements and error correction.

Third, the target processor being a core 8051, it was expected that a lot of target-specific language constructs (as presented in §III) would not be supported by the previously used commercial tools. The open source environment being highly customisable, it was possible to create extensions which would effectively model these particularities, and then be translated to code that would properly run on the target.

Fourth, the open source environment allowed us to master the complete chain, from the modelling to the code generation. Where it was generally required to submit multiple requests for deviation against standard coding rules in previous projects, the TARANIS environment allowed us to be compliant to these rules, by adapting the code generation mechanisms.

Finally, the mastering of the toolchain also allowed a very fast turnaround in terms of toolchain maintenance. Inevitably, this project being the first to use this new approach, there were multiple points in time where corrections and adaptations were required in the environment. In most of the cases, only a few hours passed between the reporting of the problem to the modification being available. This is something that would be totally unexpected from a commercial tool. And while the occurrence of such a problem is very less likely with a mature tool, knowing that blocking issues related to software development environments can be solved within days is one less risk factor for the project.

VI. CONCLUSION

As a conclusion, the development and the design of the TARANIS MIU software based on Eclipse modelling techniques was a real technical challenge in the field of critical software for space applications. Along the design and thanks to Eclipse Papyrus, Spacebel found the adequate solution.

At this time, the flight candidate software version has been delivered to CNES and is submitted to intense testing. Up to now, the quality level of the code and the documentation produced using this process did not show any inferiority with respect to those produced using commercial tools in their design process. On the contrary, Spacebel was able to produce software adapted to slightly exotic conditions, while completely remaining within the operational limits of our toolchain, and not relying on any workaround to do so.

Given the success of this pilot project, the Eclipse and Papyrus based toolchain is now being deployed on most embedded-C projects at Spacebel.

VII. REFERENCES

[1] ECSS European Cooperation for Space Standardization, Software, ECSS-E-ST-40C, 2009

[2] Eclipse Papyrus, https://eclipse.org/papyrus/

[3] ECSS (European Cooperation for Space Standardization), Software Engineering Handbook, ECSS-E-HB-40A, 2013.

[4] J. Hardy, UML Generator for Embedded C, 2015

[5] Eclipse UML generators project, https://eclipse.org/umlgen/