DOMAIN SPECIFIC LANGUAGES FOR EFFICIENT SATELLITE CONTROL SOFTWARE DEVELOPMENT

Andreas Wortmann, Martin Beet

OHB System AG, Universitätsallee 27-29, 28359 Bremen, Germany,
Email: andreas.wortmann@ohb.de Tel.: +49 (0)421 2020-9815

ABSTRACT

We present the motivation and an approach for the efficient development of satellite control software (flight software, onboard software) based on domain specific languages. Significant technological advances in the field of language workbenches have enabled us to develop extensions to the C programming language specific to the needs of satellite flight software. The approach is very promising as it combines the flexibility and efficiency of the C language with high-level abstractions known from modeling-tools and allows for additional adaptation specific to the space domain.

1. INTRODUCTION

Inefficiencies identified in earlier projects give reasons for improvement in various fields, including everything from tools and processes to code reuse. Stepping back and reassessing the actual needs while observing the current developments of embedded software technology and the scientific community has brought up the idea of developing a domain specific language (DSL) that perfectly matches the demands of satellite control software.

The intention behind DSLs is to concentrate on the essence of a domain and introduce first-class programming language constructs with semantics closely related to that specific domain. Separation of concerns is achieved by introduction of various (composable) DSLs, each focusing on independent concerns. This is combined with the idea of model based development, where a model serves as (single) source of information for many derived artifacts including the executable, the documentation and analyses etc. With the tools used the boundary between the model and the implementation becomes increasingly fuzzy. Analysis and visualization techniques that are usually applied to the model become applicable to the entire implementation. Additionally, implementation-level code refers directly to elements usually regarded as part of the model.

Essentially, a single artifact (which is both, the model and the implementation) is used to derive all required representations, including C code, documentation, and configuration.

The following steps are taken to achieve this improvement:

- identify the key issues that result in inefficiency
- identify the essential needs of the application domain
- identify the technology for improvement
- a prototypical implementation to prove the idea

They are described in the subsequent sections.

2. KEY ISSUES IDENTIFIED

The major fundamental issues identified in recent projects are related to both, tooling and process. Some of these are closely related to the selected approach, leaving it impossible to mitigate within an ongoing project.

Although tooling and processes are interrelated to some degree by nature, some specific examples are discussed in the following distinct subsections.

2.1. Tooling

Model-based design tools such as IBM Rhapsody, Enterprise Architect, Sirius and others define a model by composition of UML elements, creating structure and behavior diagrams. These are enriched with manually written implementation code in a “regular” programming language such as C or Ada. Structural and skeleton code is generated from the UML model and manually written code is inlined. This duality of inline code and the model specifically makes the implementation hard to maintain, debug, document, review and analyze as both parts are only loosely coupled. Specifically, they are maintained with two distinct tools.

Software unit tests are developed and executed using respective tools such as Cantata++ or IBM RTRT. These test supporting tools in general are not directly interfaced with the UML model, thus they can't draw any information from the model that would support test definition, quality, evaluation and reporting.

Software validation tests, as well as satellite validation, AIT and operational procedures, are developed technically independent from the satellite control software. As a consequence test coverage is hard to determine and changes of the software are not directly visible to the various stakeholders. Maintaining tests, AIT and Ops procedures requires elaborate change processes including iteration cycles and manual review.

2.2. Processes

Information that is commonly used onboard as well as on-ground is maintained in a central data repository, the Satellite Database (SDB). This central repository is used as an external model to the flight software. Code and configuration data is generated from the SDB and linked with the flight software in a later step. Consistency of the SDB content and the software technically can't be assured until the entire software is compiled, linked and validated. As this is rather late in the process, bugs are detected late and fixes are costly.

In order to be self contained and complete, the Satellite-to-Ground Interface Control Document (SG-ICD) contains – among other things – the definition of valid commands, parameters, arguments, attributes and possible failure codes, all of which are relevant to the software development. In order to ensure full validation, the SG-ICD commonly is made applicable as a source of requirements. Since large parts of the SG-ICD including the failure codes are driven by implementation details this may result in a circular dependency, calling for requirement change requests until very late in the process.

3. ESSENTIAL NEEDS OF THE DOMAIN

The high-level essential needs that a satellite control software development and maintenance process has to fulfill are the "usual suspects" of optimizing code quality, development time, cost-efficiency and V&V effort while allowing late requirement changes and early prototyping (Faster, Later, Softer). When focusing on the technical details of how to achieve these goals, the arguments discussed in the following are split into two groups: design and development considerations and generic building blocks of the software, called software elements.

3.1. Design and Development Considerations

As a developer you are generally driven by schedule and cost and at some point in the project you just have to "get it done". As a consequence, a preferred way is to base a software design on experience and heritage from earlier systems and consolidate these with the customer needs. On the other side you have to develop according to (external) **guidelines**, **nomenclature** and **processes**, as for example MISRA [11], OSRA [7], SOIS [12], PUS [8] as well as company coding guidelines. Introducing new rules and standards naturally requires some (learning) effort and may lead to hesitance or opposition. To successfully introduce and apply such standards, as much of the theoretical background imposed as possible needs to be made transparent to the user. Furthermore, user-optimized IDE support with respect to the applicable standards is appreciated.

The implementation must comply with **quality criteria**, such as those imposed by the ECSS. This generally includes software unit testing, validation tests, conformance to code metrics and a schedulability analysis. The software architecture and implementation should be designed with respect to test and analysis. This facilitates achieving the required quality standards and reduces the effort needed to obtain the expected level of quality.

A precise and detailed design **documentation**, a user manual, a satellite-to-ground ICD and other documents need to be provided and maintained. It is crucial that these documents are always consistent with the actual implementation. Well-considered structuring and writing and a large share of auto-generation is needed in order to obtain high-quality documents. Maintaining significant parts of the documentation as close to the actual implementation as possible and generating figures and tables directly from the implementation code supports this process.

3.2. Software Elements

A number of software building blocks, elements and patterns are very common to any satellite control software. What follows is a non-comprehensive list of such elements.

Component-based software design is state-of-the-art and eases the test, validation and reuse of the software because components are self contained with well-defined interfaces. Recent initiatives including the OSRA and SOIS are all based on components. Components comprise **attributes** and define **functional behavior** that may be exposed and provided to other components and/or ground.

Functionality commonly is described as a **sequence of actions**. Such sequences can be invoked internally from the software as well as via telecommand from ground. In general, sequences depend on functionality spread all over the software and multiple independent or inter-related sequences may be executed concurrently. Actions (especially when accessing hardware) may be subject to hard real-time constraints or may require synchronization with external stimuli.

Periodic (e.g. autonomous functions and checks) and **aperiodic functionality** (e.g. telecommands and other external stimuli) needs to be executed. Typically the software comprises a base frequency and a number of derived frequencies for execution of distinct and real-time sensitive repetitive functionality. Additionally, non-timing critical and sporadic functionality is executed concurrently. The task design and the scheduling scheme and algorithm is strictly designed according to the requirements of the timing critical actions and functions.

Commandability and observability of the satellite is achieved through a **satellite-to-ground interface** for telecommanding and telemetry. It is typically based on the PUS [8] but proprietary protocols or (in the future) CCSDS MO [9] are alternative options. Much information, such as the types and numbers of the arguments, describing telemetry and telecommanding is required onboard as well as on-ground.

It has been shown to be advantageous to define modes of operation for the entire satellite control software as well as for its constituent components. A **mode** in this sense is a special component attribute used to affect the components behavior in a consistent and common way. For example telecommands may only be executed in certain modes or periodic activities are always activated in selected modes.

Last but not least a well elaborated failure detection, isolation and recovery (**FDIR**) concept must be realized supporting redundancy schemes as required to achieve the needed overall reliability. Incorporating such a concept right from the beginning, when thinking about the software architecture, is necessary but rarely done.

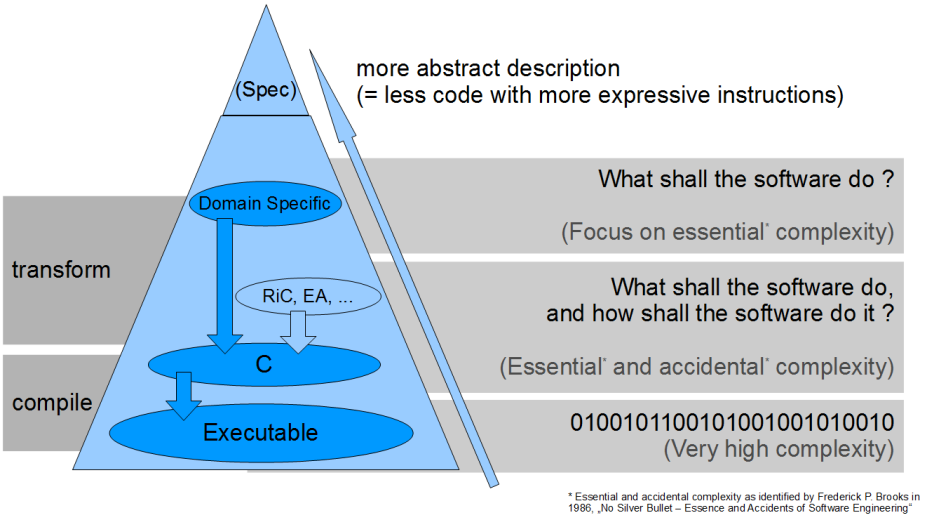

Figure 1: Pyramid of Abstraction

4. DOMAIN SPECIFIC LANGUAGES

The emerging technology of Domain Specific Languages (DSL) and their development environments, so called language workbenches, has great potential to mitigate many of the listed inefficiencies and effectively fulfill the essential needs outlined above. The essential idea is to concentrate on the specific needs of a particular domain and provide efficient means for expressing the solution of a problem in a language (or more generally: by some distinct notation) known to the domains professionals. Aspects of the implementation that are self-evident or repetitive (in that specific domain) are abstracted away. In practice the user (e.g. the implementer) does not need to keep them in mind and is able to focus on the essential objectives. The high-level and domain specific abstract description of the software objectives is automatically transformed into various concrete representations including implementation code, documentation or input for analyses tools. Essentially the implementation is a model comprising everything.

The idea behind this approach is fully in line with statements by F.P. Brooks in his paper "No Silver Bullet – Essence and Accidents of Software Engineering" [5]. The idea of introducing a compiler for writing computer programs in a human-readable language rather than in a series of '0's and '1's is merely taken to a level where even more accidental complexity is removed from the manual work share. This is achieved by shifting more knowledge about the applications domain (which is how to operate a satellite) into the language semantics.

Also the idea addresses the findings and proposals of the NASA Study on Flight Software Complexity [10].

Once a DSL has been developed for the specific domain of satellite control software, the gap between specification and implementation in our projects is narrowed significantly. Fig. 1 illustrates the great advantages of this increase in abstraction.

Basically, the software implementation focuses on the essential functionality rather than on the details of realization. A well-designed DSL allows describing what the software shall do without the possibility to introduce ambiguities. A subsequent transformation process generates C code (in the case of satellite control software) that is compiled to the executable using the established tool-chain.

5. MPS AND MBEDDR

Jetbrains' **Meta Programming System** (MPS) [1] is a language workbench [13] (DSL Development Environment) that is open source and freely available. It applies projectional editing. This overcomes the limits of language parsers and allows editors to include tables, mathematical symbols and graphical diagrams. The user directly modifies the abstract syntax tree (AST), respectively the implementation model, which is projected onto the screen, see Fig. 2.

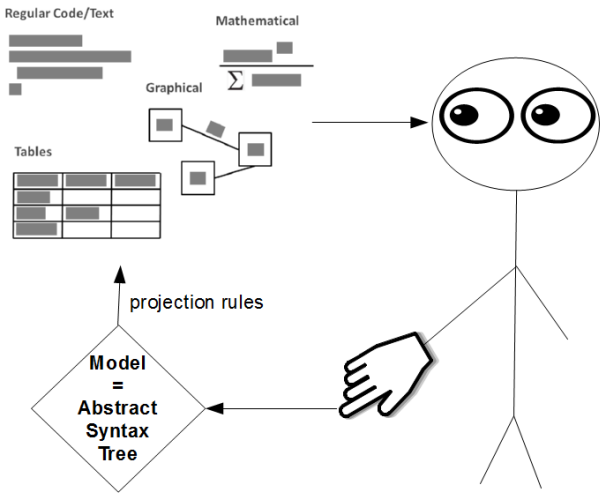

Figure 2: Projectional Editor

Different visualizations can be selected, depending on the current needs, all representing (parts of) the same model. Since a parser is not engaged, languages are composable by construction and can be arbitrarily mixed in an implementation.

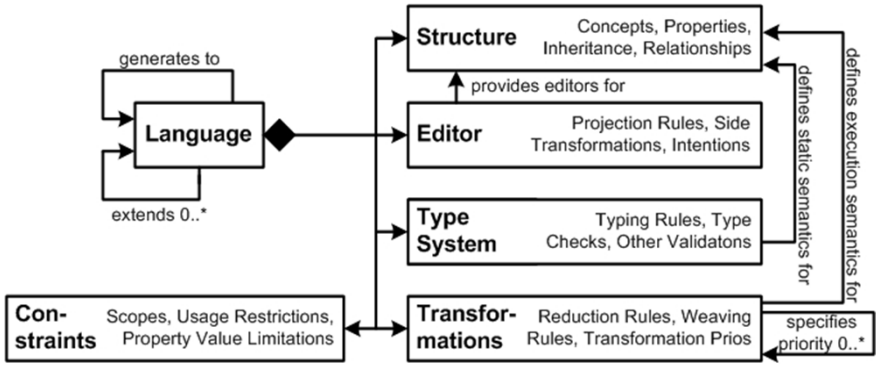

Figure 3: Language Definition

Languages in MPS consist of an abstract syntax (the structure), a type system, a set of constraint rules, (multiple) editors and model-to-model as well as model-to-text transformations, compare Fig.3. As the MPS framework is open for extension, it can be easily augmented to interface with various external tools. For example, analysis tools are directly integrated and their results are reported in the editor. The AST is the implementation model which is transformed via a number of (model-to-model) transformation steps into a textual representation. In the case of satellite control software this could be C code for further processing by the compiler. During the transformation process available MPS languages can be used and the entire model is available for queries for efficient transformation results.

The **mbeddr** project [2] is based on MPS. It provides a language in MPS that looks exactly like the C language, has the same semantics and transforms into C code for the compiler to process. The advantage of having the C language available as an MPS language (called mbeddrC) is that it is now possible to extend the language with domain-specific abstractions, while at the same time being able to write low-level C code when necessary. A number of such extensions including finite state machines, documentation, requirements tracing, model-checking, visualizations etc. have been developed as part of the mbeddr project. The mbeddr stack of extensions is shown in Fig. 7. mbeddr is available as open source software, professionally maintained by Itemis AG [6] and has been successfully applied within commercial projects as reported in [4].

6. A PROTOTYPICAL IMPLEMENTATION

Based on the technology provided by MPS and mbeddr, a set of DSLs is currently being developed by OHB in the scope of R&D activities. The DSLs capture and efficiently implement the crucial elements of satellite control software. Redundant and repeatedly applied aspects are captured in higher level abstractions (first-class DSL constructs). The DSLs are modular in terms of the addressed concerns. This, together with a fully integrated development environment providing state-of-the-art user guidance and team development support, results in a highly efficient approach to software development.

The DSL design is mainly influenced by heritage from earlier projects carried out at OHB (Small Geo Platform and Galileo), the ESA SAVOIR initiative OSRA [7] and the ECSS Packet Utilization Standard (PUS) [8]. Programs written in the DSLs are compliant to the ECSS standards and quality requirements.

According to the list of software elements as presented in section 3.2 the mbeddrC language is extended with higher level concepts. Appropriate structure, editor rules, type system and constraints are defined. Finally, transformation rules are defined for generating C code and documentation.

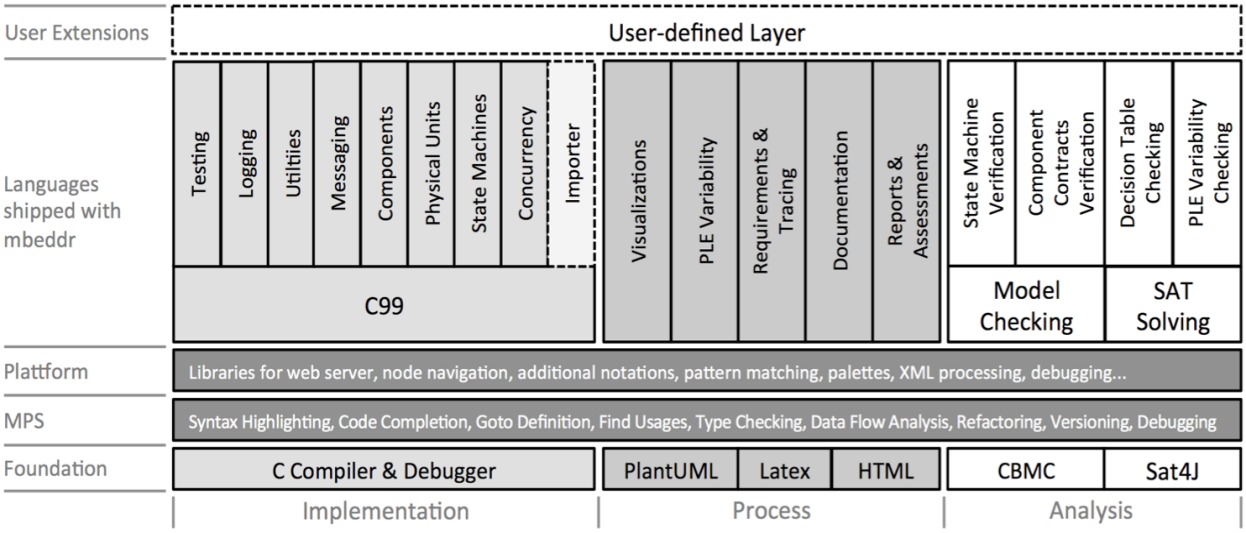

Figure 4: mbeddr Stack of Extensions

```
#constant TEMP_BUFFER_SIZE = 10;

TACQA = Instance of TemperatureAcquisition with mnemonic tail A and the Numeric Id 350
  [SENSOR = SensorA]
TACQB = Instance of TemperatureAcquisition with mnemonic tail B and the Numeric Id 351
  [SENSOR = SensorB]

[Layer: EPL.PFD (Platform Devices)]
Component TemperatureAcquisition with Base Mnemonic: TACQ
  Short Description: acquisition of temperatures
  Description: The components acquires the measurements of an assigned set of thermistors
{
  Attribute (hidden) int32/rawTemp/[TEMP_BUFFER_SIZE] MEASURED = <no init> ; // measured raw values
  Attribute (hidden) uint32 ACQCNT = 0 ; // index for filling data acquisition buffer
  Attribute (readwrite) tempSensor SENSOR (Id= 2 ) = <no init> ; // selected sensor for this component

  ModeChart TCSACQ (Id= 3 ) initial = OFF {
    Trigger tcsAcquisition
    Mode OFF {
      << ... >>
    }
    Mode ON {
      entry { ACQCNT = 0; }
      on trigger tcsAcquisition {
        // measure a value
        MEASURED[ACQCNT] = readTemperature(SENSOR);
        ACQCNT = (ACQCNT + 1) % TEMP_BUFFER_SIZE;
        // calculate average of the @top(TEMP_BUFFER_SIZE) latest measurements and convert to °C
        PUS150.AVTEMP = convert[ Σ_{idx=0}^{TEMP_BUFFER_SIZE-1} (MEASURED[idx]) / TEMP_BUFFER_SIZE -> °C];
      }
    }
  }

  Activity startAcquisition with Nummeric Id 1
  ... { TCSACQ.setMode(ON); }

  Activity stopAcquisition with Nummeric Id 2
  ... { TCSACQ.setMode(OFF); }
} Component TemperatureAcquisition
```

Figure 5: Temperature Acquisition Component

```
PUS150 = Instance of ThermalControlSystem [<< ... >>]
[Layer: AL (Application Layer)]
Component ThermalControlSystem is Service 150 has M
  Short Description: simplified thermal control
  Description: The component implements a very simp
{
  // thermal control
  ModeChart TCSCONTR (Id= 0 ) initial = OFF {
    Trigger tcsControl
    // thermal control is inactive
    Mode OFF {
      << ... >>
    }
    // thermal control is active
    Mode ON {
      // disable heater
      exit { switchHeater(OFF); }
      // periodically triggered for altering the
         heater power state according to the
         measured values
      on trigger tcsControl {
        heaterState hCmd;

        hCmd =                                  true otherwise UNCHANGED;
               PUS150.AVTEMP <= PUS150.LOTH     ON
               PUS150.AVTEMP >= PUS150.UPTH     OFF

        switchHeater(hCmd);
      }
    }
  }

  Attribute (readonly) int32/°C/ AVTEMP (Id= 1 ) = <no init> ; // average of measured temperature
  Attribute (readwrite) int16/°C/ UPTH (Id= 2 ) = 15 °C ; // upper (switch off heater) threshold
  Attribute (readwrite) int16/°C/ LOTH (Id= 3 ) = 5 °C ; // lower (switch on heater) threshold

  Activity enableTcs with Nummeric Id 1 is commandable by TC(150,1)
    Short Description: enable thermal control
    Description: The thermal control heats the system if it is too cold. The switching histeresis can be configured.
    Constraints:
      0: TCSCONTR.inMode(OFF ) // switching on is possible only if the TCS is off
    In-Parameter:
      int16/°C/ upperThreshold: constrained :  <no constraint> // upper switching threshold
      int16/°C/ lowerThreshold: constrained :  lowerThreshold < upperThreshold // lower switching threshold
      component<TemperatureAcquisition> acq: constrained :  <no constraint> // acqusition component instance to use
    {
      REQUEST acq.startAcquisition ( << ... >> ) --> ( << ... >> )
        on error do nothing special
        on error abort
      UPTH = upperThreshold;
      LOTH = lowerThreshold;
      DELAY for 10 s
      TCSCONTR.setMode(ON);
      TELEMETRY (150,11)
        Description: Report switching on in a dedicated packet that reports the initial temperature.
        [initialTemp : int32/°C/ = PUS150.AVTEMP // initial temperature when starting thermal control ]
    }

  Activity disableTcs with Nummeric Id 2 is commandable by TC(150,2)
    Short Description: disable thermal control
    Description:
    Constraints:
      0: TCSCONTR.inMode(ON ) // switching off is possible only if the TCS is on
    In-Parameter:
      << ... >>
    {
      TCSCONTR.setMode(OFF);
      REQUEST TACQA.stopAcquisition ( << ... >> ) --> ( << ... >> )
        on error do nothing special
      REQUEST TACQB.stopAcquisition ( << ... >> ) --> ( << ... >> )
        on error do nothing special

    }
} Component ThermalControlSystem
```

Figure 6: Thermal Control Component

Components as basic structural element and their instances are first-order constructs. Besides other elements the components contain attributes, mode-charts and activities. Fig.5 and 6 show screenshots of an implementation based on DSLs extending the C language. Everything that does not look like C belongs to a language extension. The used elements are briefly described in the following.

Fig.5 depicted the implementation of component `TemperatureAcquisition` (1) and two instances of this component (2). The component is assigned a mnemonic (`TACQ`) that is reused and augmented with a mnemonic tail (`A` and `B`) by the instances. The resulting component instances' mnemonics (`TACQA` and `TACQB`) are derived and projected into the editor. For addressing purposes unique numeric Ids (`350` and `351`) are provided with the instances. The component implementation (1) comprises mandatory (short and regular) description fields which are required for the generated documents. The component is assigned to the `EPL.PDF (Platform Devices)` layer which is one of several predefined layers and has the same intent as that defined in the OSRA (3). The layer technically (enforced by the editor) restricts access to other layers' component instances.

Furthermore the component defines three attributes (4), by convention (and technically enforced by the editor) they are all capital letters. They act as variables local to the component instance which may be configured to be accessible by ground via a dedicated Service. For this purpose, attributes are marked either 'hidden', 'readonly' or 'readwrite' and – if not hidden – are assigned a numerical Id for identification via the space-to-ground protocol. Note that the entire data pool maintaining the attribute values and their access by other components and via telecommands from ground (PUS 3 and PUS 20 services) is not mentioned here. It is abstracted from the DSL-level implementation and generated during model-to-model transformations. Thus, the implementation only focuses on the essential concern, which is the provision of an attribute. Like a regular C variable declaration an attribute is assigned a data type and may be initialized. The editor requires the implementer to add documentation. An attribute can be initialized differently for each component instance. In the example the attribute `SENSOR` (which is of enum type `tempSensor`) is initialized differently for both instances. Together with the instance's mnemonic the attributes are uniquely addressed in a hierarchical way using a dot expression syntax (5). In the example the attribute `AVTEMP` from instance `PUS150` is accessed (see Fig. 6 for its definition).

The yellow annotation `/rawTemp/` (6) to the datatype of the array is part of a language extension that allows associating a physical unit to the values carried in this variable. Here the value represents a raw value read from the thermal sensor. In Fig. 6 attributes are associated with centigrade (16).

The `ModeChart` statement (7) implements a simple kind of state-chart efficiently capturing the idea that every component may reside in an operational mode. This mode is changed by dedicated statements invoked in the activities (7). The ModeChart in Fig.5 defines two modes (`OFF` and `ON`) and a trigger. Mode `OFF` does not define any behavior, but mode `ON` has an entry action and an action that is associated with the trigger `tcsAcquisition`. When entering this mode the (hidden) attribute ACQCNT is initialized. When the trigger is invoked (by a periodical task/thread not shown here) a temperature value is read from the sensor and an average of the last 10 measurements is calculated. While invoking a function for reading the temperature value, calculating the index and accessing the array for storing 10 values is regular C code the calculation of the average temperature uses an extension that provides mathematical symbols (8). This extension allows

$$\sum_{idx\ =\ 0}^{TEMP\_BUFFER\_SIZE\ -\ 1} (MEASURED[idx]) \ /\ TEMP\_BUFFER\_SIZE$$

to be an expression that is transformed into a respective loop for doing the calculation. Projectional editing allows the mathematical symbol to be made visible in the editor. The mathematical formula is wrapped with a `convert[ … → °C]` statement. This is part of the language extension that provides physical units as used at (6). It converts the value from `rawTemp` to `°C` as defined for example in the following sketched conversion rule:

```
exported unit °C := _ for degree Celsius
exported unit rawTemp := _ for raw Temperature value
#constant PT1k_A2 = 0.00000116332;
#constant PT1k_A1 = 0.2316700824;
#constant PT1k_A0 = -243.3112318176;
exported conversion rawTemp -> °C {
  val as int32 -> ( val² * PT1k_A0) + (val * PT1k_A1) + PT1k_A0
}
```

If expressions are assigned to variables with a non-matching physical unit the typing rules of the language extension report the error in the editor directly at implementation time.

Functionality is implemented by `ActivityS` inside a component. The activities shown in Fig.5 are rather simple and their representation in the editor is folded for brevity (9).

Fig.5 shows a simplified component for acquisition of temperature values and calculating an average. Two instances of this component are realized. The component shown in Fig. 6 uses these instances, it realizes a very simple thermal control system that is able to use either one of the component instances for acquisition.

The component `ThermalControlSystem` as shown in Fig. 6 implements a service according to the PUS standard. The component statement is marked accordingly and is provided with a PUS type (`150`). In order to make this more visible in the editor, the background is colored yellow. Services are only instantiated once and their mnemonic is `PUSxxx` where `xxx` is the assigned PUS type (10). The component is assigned to the `AL (Application Layer)` and as such is allowed to access the `TemperatureAcquisition` instances assigned to the `Platform Devices` Layer (3).

The `ThermalControlSystem` component defines a ModeChart. It is visualized in Fig. 7. The visualization is generated directly from the implementation by

invoking an external tool from within the IDE. It – by nature – is always consistent with the implementation, such that code review actually can be carried out using the printed documentation.

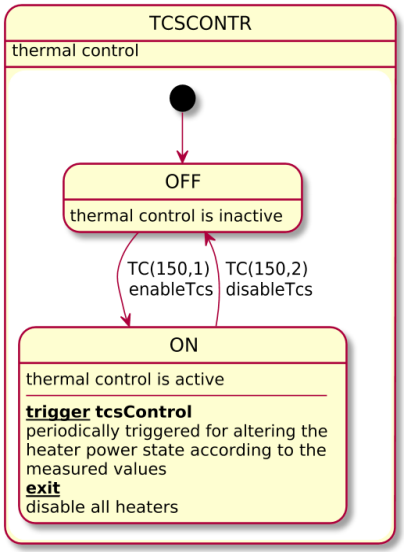

Figure 7: ModeChart

In mode `OFF` the control is inactive. In mode `ON` it reacts on the trigger `tcsControl`, which is periodically invoked in the context of a periodic task/thread. In this trigger action the commanding for the heater lines is determined by comparison of the average temperature `PUS150.AVTEMP` with the thresholds `PUS150.LOTH` and `PUS150.UPTH`. The comparison and the resulting heater command is realized using a language extension that provides a decision table (11). Depending on the boolean expressions on the rows and columns a value is selected and returned. The shown example is a very small decision table. The mentioned component attributes (12) are associated with the physical unit `°C` ensuring that the check is carried out on comparable values.

The `ThermalControlSystem` service defines two activities that define functionality (13). Each activity is assigned a numerical id, name and description fields. As there is only a single instance of this component (service) its activities can be marked a telecommand. The telecommands PUS compliant type identifier is derived from the services numerical id by definition and the subtype identifier is the activities numeric id (14).

An activity may be constrained by a set of boolean expressions (15), only if all of them evaluate to true when invoked, the activity is actually executed. If an invoked activity does not fulfill a constraint a failure code is returned instead. The failure code is unique and derived from the numerical ids assigned to the component instance and the activity according to the structure of the software. In the example the mode of the ModeChart `TCSCONTR` is required to be `OFF` or `ON`, respectively. An activity can define input and output parameters (15), telecommands are restricted to input parameter due to limitations of the PUS. Such parameter are typed and mandate a description field. In-parameters may define a constraint, for validity checking. The in-parameter constraint is a boolean expression and acts similarly to the activities' constraint. In the example activity shown, `TC(150,1)enableTcs` is only executed with the components mode being `OFF` and the passed in-parameter `upperThreshold` is larger than the passed in-parameter `lowerThreshold`.

The invocation of activities is according to the MO MAL interaction pattern [9]. Based on an execution framework middleware, the activities are invoked asynchronously. The adaptation to the middleware is generated as part of the transformation process. Depending on the middleware the software is expected to be operated with, different sets of transformation rules may be applied. The implementation itself is designed independently of the middleware. In Fig. 6 the activity `enableTcs` uses the REQUEST pattern to invoke the `startAcquisition` activity of the addressed `TemperatureAcquisition` component instance (17). The REQUEST pattern allows in- and out-parameters to be transmitted and the respective statement may specify a block of code that is to be executed if a failure indication is returned. The control flow requires the invoked activity to complete execution prior to the invoking activity to continue. In contrast, the SEND pattern asynchronously invokes an activity and does not wait for any acknowledge. Fig. 8 shows a sequence diagram visualizing the telecommands implementation. Again, this picture is generated from the implementation using an external tool from within the IDE.

The telecommand (TC) Dispatcher is not shown in the code. It is aware of all telecommands defined in the software by querying the implementation. Via model transformation the needed telecommand dispatching harness is generated, enabling invocation of the activity via TC packets received.

The activity delays enabling of the thermal control by 10 seconds using the `DELAY` statement (18), which transforms into respective delay functionality provided by the framework. The last statement of the telecommand shown in Fig. 6 is the `TELEMETRY` statement (19). It defines a telemetry packet (again,

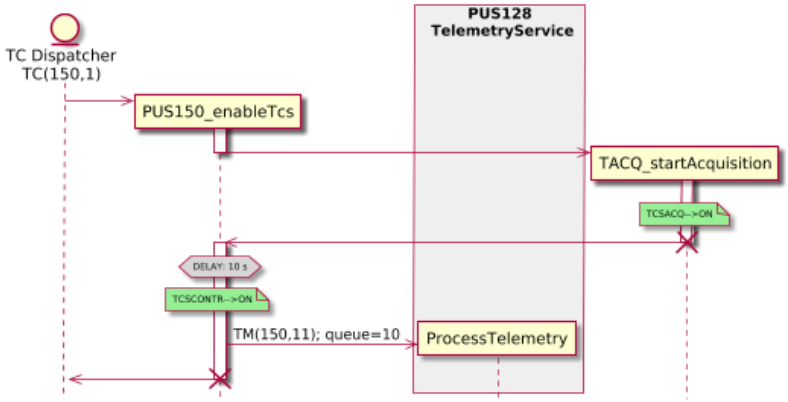

Figure 8: Sequence Diagram

according to PUS) by specifying a mandatory description, a subtype (the type is inherited from the embracing service) and a list of typed parameters. Only this statement defines the telemetry packet, other occurrences merely refer to it. During code generation this statement is transformed into an activity invocation according to the SEND pattern. The activity responsible for preparing and transmitting a telemetry packet is invoked as shown in the sequence diagram in Fig. 8. The corresponding arrow in the figure is annotated with "`queue=10`". This indicates that the activity provides a queue, allowing different parts of the software to invoke the activity and respective invocations being queued. Again, the concrete realization of the queue is not part of the implementation, it is part of the transformation rules defined in the DSL statement with respect to a selected middleware.

From the implementation not only executable code is generated. As briefly summarized in Fig. 9, other aspects are covered also. A dedicated DSL allows efficient requirement tracing, state-of-the-art software revisioning enables efficient team collaboration and issue tracking. The implementation serves as source for documentation generation and the configuration of the (ground-) operator software system. Dedicated DSLs are developed for the inclusion of unit tests and analyses. While mbeddr already provides means to show formal properties of certain state machines and decision tables, we are currently working on a DSL linking the implementation with static worst case execution time (WCET) and schedulability analysis.

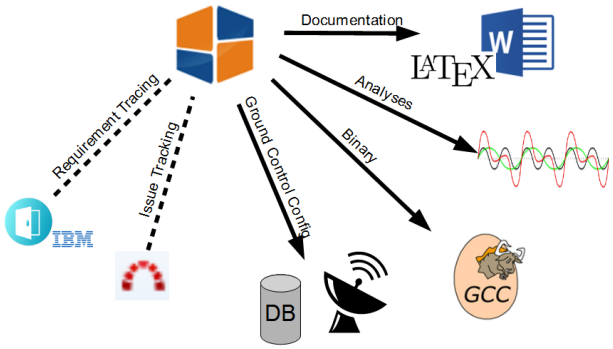

Figure 9: Linked and Generated Aspects

As the implementation contains the entire design information and arbitrary queries can be executed, it is feasible to simplify the data dependencies and as a consequence the overall workflow. Circular and overly complex process-related dependencies with respect to the Satellite-to-Ground ICD and the Satellite Database as described in section 2.2 are simplified.

7. SUMMARY AND OUTLOOK

A significant increase in development efficiency and quality is achievable when the focus of work concentrates on essential aspects and recurring efforts are abstracted.

In recent years the technology of domain specific languages and language workbenches has reached a level of maturity that allows its application in the field of embedded, real-time, mission-critical flight software of spacecraft. In addition to what is accomplished by libraries, language extensions provide custom syntax and type systems as well as the possibility to introduce static error checking. Context sensitive model-to-code transformation allows for optimized code generation resulting in run-time and memory efficient code. Compared to other concepts the approach is non-disruptive as it builds upon the C language and experienced developers feel comfortable. Established tools and analyses can be introduced smoothly. The projectional editing and introduction of higher level abstractions cause the concepts of modeling and implementing to become somewhat blurred, taking advantages from both.

The work carried out and prototype developed has shown that the approach and the tooling is very promising. It bears great potential for improving current workflows. Up to the present and while significant elements and aspects of satellite control software have not been tackled, the full advantage has not been drawn from the approach. Nevertheless, the DSL based approach of embedded software development has been reported very successful in a commercial environment [4]. It is expected to achieve improvement results of similar magnitude and quality.

The next step in evaluation is to cover the entire life-cycle (requirements engineering, architecture design, implementation, validation and operation) and all major functional aspects of in an exemplary satellite control software implementation.

ACKNOWLEDGMENTS

This work is based on the technology provided by Jetbrains MPS [1] and the mbeddr project [2]. It has been carried out in the scope of the ARTES 11 Subelement 3 project under ESA contract number 20619/2007/F/W.

REFERENCES

[1] Jetbrains MPS, www.jetbrains.com/MPS
[2] The *mbeddr* project, www.mbeddr.com
[3] M. Völter, *DSL Engineering*, www.dslbook.org, 2013.
[4] M. Völter et.al., *Using C Language Extensions for Developing Embedded Software: A Case Study,* OOPSA2015
[5] F.P.Brooks, *No Silver Bullet – Essence and Accidents of Software Engineering*, Proceedings of the IFIP Tenth World Computing Conference, 1986
[6] Itemis AG, www.itemis.de
[7] SAVOIR-FAIR–OSRA, *Onboard Software Reference Architecture*, TEC- SWE/09-289/AJ
[8] ECSS-E-70-41C, *Packet Utilization Standard*, 2014
[9] CCSDS 521.0-B-2, *Mission Operations– MAL*, 2015
[10] D.L.Dvorak et al., *NASA Study on Flight Software Complexity*, Final Report, March 2009
[11] MISRA, www.misra-c.com
[12] CCSDS 850.0-G-2, *Spacecraft Onboard Interface Services*, December 2013
[13] Language Workbench Challenge, www.languageworkbenches.net