

# End-to-end observatory software modeling using domain specific languages

José M. Filgueira, Matthieu Bec, Ning Liu, Chien Peng, José Soto  
Giant Magellan Telescope Project, 251 South Lake Ave., Suite 300, Pasadena, CA, USA 91101

## ABSTRACT

The Giant Magellan Telescope (GMT) is a 25-meter extremely large telescope that is being built by an international consortium of universities and research institutions. Its software and control system is being developed using a set of Domain Specific Languages (DSL) that supports a model driven development methodology integrated with an Agile management process. This approach promotes the use of standardized models that capture the component architecture of the system, that facilitate the construction of technical specifications in a uniform way, that facilitate communication between developers and domain experts and that provide a framework to ensure the successful integration of the software subsystems developed by the GMT partner institutions.

**Keywords:** GMT, software engineering, DSL, Agile, model-based development, lean

## 1. INTRODUCTION

The GMT Software and Controls System encompasses the software and hardware components necessary to control and monitor the GMT optical and electromechanical subsystems and to safely and efficiently operate the GMT observatory (Figure 1-1). Its design is driven by a set of general guidelines: to use industry components and standards that improve the cost-effectiveness of the system; to design an architecture around well-established practices and design patterns; to validate the technical platform and architecture by prototyping and incremental delivery; to use a model-based development approach integrated with an Agile and lean based management process; and to efficiently support and collaborate with the parties involved in the development of the different subsystems.

The GMT Software and Controls System includes a significant number of Device Control Systems (DCS). Some of them are developed in house, while others are procured by the partner institutions or commercial vendors located around the world. In order to facilitate the specification, development, integration, and standardization of common design aspects, DCS subsystems are organized in a modular reference architecture (Figure 1-2) that improves the internal consistency of subsystems, minimizes the interfaces between them and facilitates the composition of individually tested subsystems. The high level view of this architecture shows the following main elements:

- GMT SWC Core Systems – Includes all the hardware and software required to operate the observatory and telescope as an integrated system and to coordinate all the DCSs. It comprises:
  - Observatory Operations System – Provides the functionality to efficiently operate the observatory (e.g. observing tools, scheduler)
  - Observatory Services – Include common infrastructure observatory services (e.g. telemetry service, configuration service)
  - Telescope Control System – Includes the system level telescope control functions (e.g. wavefront control, pointing).
- Controlled Subsystems – Control relevant part of a system to achieve the specified control objectives. This includes the control system and the controlled plant (e.g. M1).
  - Device Control Subsystems – Includes all the hardware and software required to control a Subsystem Controlled Plant. DCS include a set of Device Control Packages and may include a set of Operation Support Packages.

- Device Control Packages – Provide functions to control and supervise the Subsystem Controlled Plant (e.g. motion control functions).
- Operation Support Packages – Provide the functions to operate a Controlled Subsystem integrated with the rest of the observatory. (e.g. visualization or calibration functions, telemetry integration).
- Subsystem Controlled Plant – Physical system, or one of its parts, which is the target of the control problem. Includes sensing and actuating devices. (e.g. M1 support system)

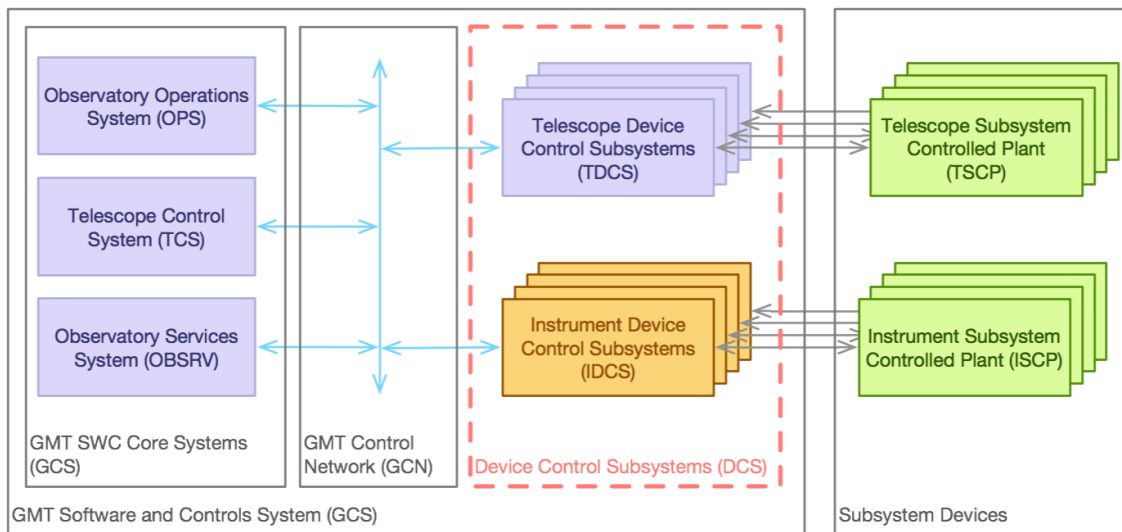


Figure 1-1. Overall view of the GMT software and controls system. Telescope and Instrument DCS account more than 30 subsystems

As an example, during the execution of an observation, the DCS of instruments, telescope and adaptive optics are coordinated by the software core systems (e.g. operation sequences executed by the Sequencer, high order control loops executed by the wavefront control system) in order to execute automated observation or calibration operations. The instrument data pipeline packages are integrated with the core Data Processing System allowing quick look and data products management to be operated in a uniform way across the observatory. Analogously instrument specific user interface panels and observing tools are integrated into the core Observing Tools to ensure consistent and seamless implementation of observatory workflows.

In order to address the challenges associated with the distributed development of complex distributed control systems, an end-to-end software modeling strategy has been designed. The following sections describe how this strategy is being used to standardize terminology, to describe the architecture of the system and the technical specification of components, and lastly to define both development and operational processes. Section 2 describes the principles of the GMT modeling framework. Section 3 presents some modeling scenarios that are being developed and some samples of how, once the architecture is captured in the semantic model, it can be used to drive many of the software development activities.

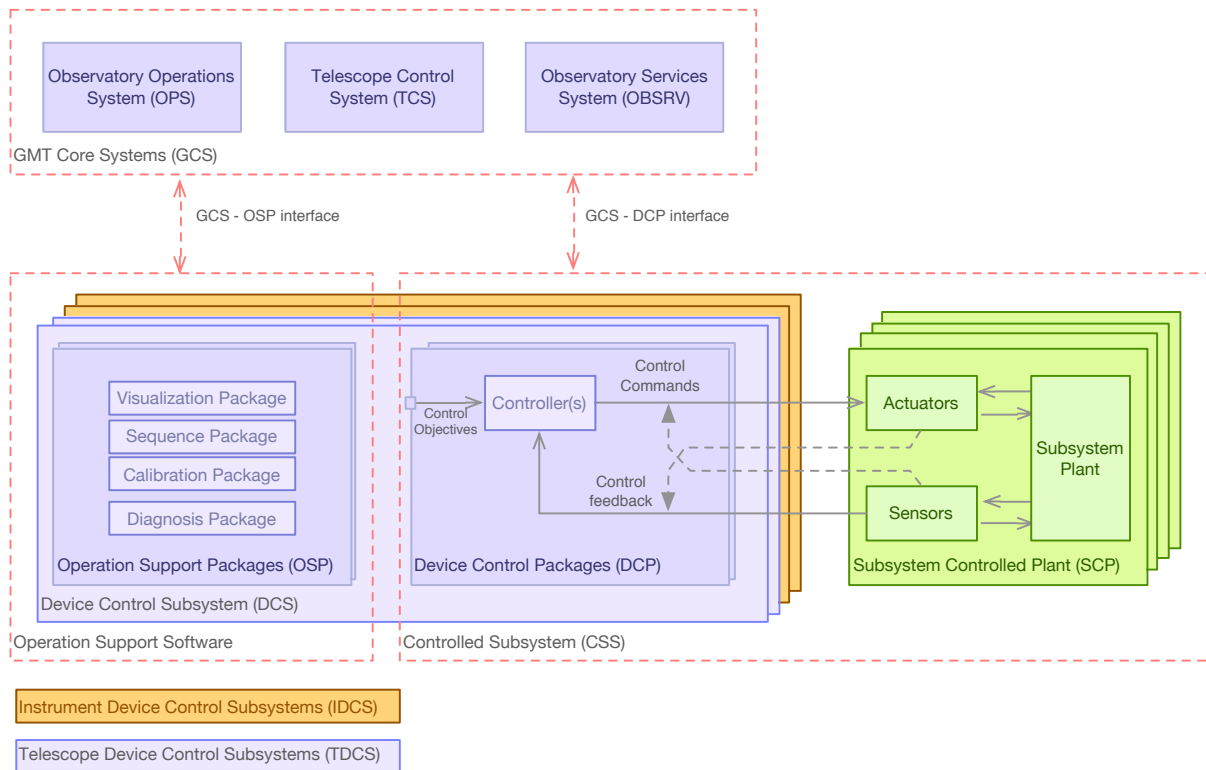


Figure 1-2. Device Control Subsystem Reference Architecture

## 2. MODELING FRAMEWORK

The Domain Engineering Process is one of the Software Reuse Processes as defined in ISO/IEC 12207<sup>1</sup>. As stated in the standard, domain engineering is a reuse-based approach to define the scope (i.e., domain definition), to specify the structure (i.e., domain architecture), and to build the assets (e.g. requirements, designs, software code, documentation) for a class of systems, subsystems or applications. The Reuse Asset Management Process and the Reuse Program Management Process complete the Software Reuses Processes. In the case of GMT software systems reuse assets are implemented as component frameworks and specification files that capture domain specific architectures and as a library of reusable components.

In software engineering, domain analysis - or product line analysis - is the process of analyzing related software systems in a domain to find their common and variable parts. This approach to domain analysis is also called model-driven engineering. Model-driven engineering<sup>2</sup> “is a software development methodology that focuses on creating and exploiting domain models (that is, abstract representation of the knowledge and activities that govern a particular application domain)”<sup>3</sup>.

The GMT domain analysis process creates a set of domain object-oriented models using domain specific languages (DSL)<sup>4</sup> that expand several knowledge domains (e.g. control<sup>5</sup>, data processing, observatory operations). The design of the GMT modeling framework fulfills several purposes:

- Provides a mechanism for defining the architecture of the system, similar to other component architecture definition languages (e.g. AUTOSAR<sup>6</sup>, AADL<sup>7</sup>). In this respect the model embodies the reference architecture of the software system and standardizes the terminology.
- Provides an infrastructure for elaborating the technical specifications of the different software subsystems

- Provides an efficient method to construct development tools that tap into the semantic model to create the necessary development artifacts in an effective manner (e.g. it facilitates the creation of integrated build and test systems). Software developers can use these models as a basis to implement their software subsystems and corresponding applications.

An end-to-end software model, based on a common specification language, provides continuity between domains and enables the reuse of tools. The use of DSLs for specifying architectures is a well-known practice<sup>10</sup>. A DSL is a computer programming language of limited expressiveness that is customized for a particular domain, so as to facilitate productivity and communication with domain experts and stakeholders. The use of a DSL provides concrete syntax and semantics that reduce ambiguities when specifying system properties, functionalities, etc., in that domain. The design of the GMT modeling framework is based on the Object Management Group Meta Object Facility's four-layered architecture. The concrete syntax is provided by an *internal* DSL, while a higher abstraction meta-model provides the semantics and well-formedness rules. Analogously, the meta model semantics are defined by an enriched meta-meta model based on the Eclipse EMF/ECore<sup>8</sup> and OMG/EMOF<sup>9</sup>.

Text-based languages provide several benefits over graphical representations<sup>10</sup>. DSLs provide a "path of least resistance" to domain engineers who may not be familiar with graphical formal notations like UML or SysML. Nevertheless, visual representations can be generated from the semantic model to communicate quantitative information if needed, such as, for example, for deriving requirements of common observatory services (e.g. telemetry bandwidth).

The use of an internal textual DSL hosted in Coffeescript greatly benefits from a wide variety and quantity of tools and libraries available in the web platform. Tools and the semantic model can be deployed either in a browser or in the server using *node.js*. An internal DSL eliminates the need to develop a scanner and a parser. However, thanks to the use of the *esprima*<sup>11</sup> Javascript parser is possible to access the Abstract Syntax Tree (AST) based on the Mozilla *SpiderMonkey* Javascript AST. Although the DSL is mostly declarative, having access to a representation of the AST enables the implementation of features, like code generation, from algorithmic specifications by means of a source-to-source compiler.

From the point of view of system engineering, the specification tree of the software and control subsystems is composed of a set of hierarchically organized System Definition Files (SDF). The use of the SDF terminology provides a more intuitive naming scheme, obviating the need for abstract concepts that are unfamiliar in other design disciplines. Software domain models are represented as a set of SDFs written in a DSL. SDFs are text files that capture the system architecture and the technical specification of software/hardware Subsystems, Packages or Components. SDFs are parsed and loaded in a semantic model database implemented in *MongoDB* and processed for consistency and completeness. SDFs are stored in the GMT *git* repository<sup>12</sup> that supports versioning, configuration control and distributed development. The SDFs of each subsystem are one of the deliverables of the DCS development phases and provide the basis for acceptance testing of subsystem components. Development workflows, which are also captured in the model, describe how to create system definition files in a consistent way through the project.

The modeling framework design is composed of a core that supports the creation of arbitrary modeling layers and provides common functionality for all modeling layers. Model elements can use inheritance to reuse and generalize common features. The framework also includes a plugin system that accesses the model elements and defines mixin methods that can be added dynamically to existing model elements. For example, some plugins provide the capability to generate several development artifacts (e.g. code, reports, visualizations, documentation, *CSV* files) by means of specialized, template-based, renderers. The modeling framework allows import/export from different formats (e.g. *JSON*, *CSV*) to enable interoperability with other engineering tool used by domain experts.

During model design, meta-model, model and runtime elements can be mixed in the same file to facilitate prospective modeling and quick prototyping. Modeling entities can be manipulated using the hosting language for efficient implementation of modeling tools. The model enables and enforces a common naming convention by way of regular expression patterns. Each model element can also define conformance rules. Well-formedness rules are pieces of code written in the host language that enforce constraints over the model elements, e.g. making sure that two components that are connected have compatible ports.

The next sections describe how a modeling strategy based on a component architecture definition language provides the mechanism to capture the architecture and technical specifications (requirements) during the domain engineering process.

### 3. MODELING SCENARIOS

#### 3.1 Component modeling

The design of the GMT software and controls system is based on a component architecture. *Components* represent the most elementary unit for the purpose of development, testing, integration and reuse. Groups of components can be connected to create composite modules that implement complex functions. Component interfaces are defined using *Ports*, which can be linked by means of *Connectors*. For example, connectors are used to (a) integrate standardized reusable control components with a given field bus configuration; (b) connect component responses with user interface components; or (c) connect components with common observatory services. Connectors are specified in the model without making any assumption of the underlying middleware used by the platform-specific implementation.

One area where component modeling is very useful is to enable the telescope to operate under multiple configurations. As described in Bernstein et al.<sup>13</sup>, the GMT system is required to operate in several different configurations during the development stages and during the subsequent operation lifecycle of the observatory. Stage 1 of the project development provides the capability to operate in seeing limited mode<sup>14</sup> with 4 Primary Mirror Segments and 4 Fast Steering Mirrors. Stage 2 provides the capability to operate in Natural Guide Star AO mode and Ground Layer AO<sup>15</sup> model. Stage 3 adds the Laser Guide Star AO mode and Wide Field correction. During the commissioning of the AO subsystems, a subset of subsystems (e.g. ASM, Instrument, Wavefront Sensors) is required to operate coordinated by the wavefront control system in the Facility Calibration System. Additionally during the normal operation phase, the Telescope top-end may be switched between the Fast Steering Mirror configuration and the Adaptive Secondary Mirror Configuration, with one of them operating on the telescope while the other operating on the Facility Calibration System. Additionally, during normal operation the inputs and outputs of the Wavefront Control System may change depending on the observing mode.

To address multiple configurations, a modular architecture that enables flexible configuration and reuse of components is created using a combination of components, connectors and ports. Connector sets are grouped using connector maps (e.g. Natural Guide Star AO or Partial M1/M2 Natural Seeing connection map). Each telescope configuration has a connection map that defines how component ports are connected to each other. The active connection map can be changed dynamically to allow the flexibility to reuse existing components under different configurations. Connection Maps are defined in the model and provide the basis to define the interfaces between Subsystems. Figure 3-1 shows the meta model specification for defining Components. *Alarms*, *monitors* or *commands* features are specialized classes of Ports.

```
MClass      "Component",
  extends:  ["SCI"]
  abstract: false
  desc:     "Basic architectural building block"
  features:
    commands: {kind: "containment", lower: 1, upper:-1, type: "Command",
               desc: "Definitions of the commands the Component accepts"}
    properties: {kind: "containment", lower: -1, upper:-1, type: "Property",
                desc: "List of configuration property definitions"}
    alarms: {kind: "containment", lower: -1, upper:-1, type: "Alarm",
            desc: "List of alarm definitions"}
    monitors: {kind: "containment", lower: -1, upper:-1, type: "Monitor",
              desc: "Telemetry monitors ports"}
    data_inputs: {kind: "containment", lower: -1, upper:-1, type: "Data_IO",
                 desc: "Data Input ports"}
    data_outputs: {kind: "containment", lower: -1, upper:-1, type: "Data_IO",
                  desc: "Data Output ports"}
    statemachine: {kind: "containment", lower: -1, upper: 1, type: "StateMachine",
                  desc: "Component state machine"}
    requirements: {kind: "containment", lower: -1, upper:-1, type: "Requirement",
                  desc: "Non feature requirements"}
    files: {kind: "containment", lower: -1, upper:-1, type: "File",
            desc: "Support files"}
```

```

MKlass      "Monitor",
  extends:  ["Port"]
  abstract: false
  desc:     "Monitors are the basic element for component telemetry.
            Monitors don't produce any direct action on the associated component"
  features:
    units:  {kind: "reference",  lower: 1, upper: 1, type: "UnitType",
            desc: "Units of the monitor"}

    type:   {kind: "attribute",  lower: 1, upper: 1, type: "BaseDataType",
            desc: "Type of the Monitor"}

    min:    {kind: "attribute",  lower:-1, upper: 1, type: "ValueType",
            desc: "Minimum value used for monitoring purposes"}

    max:    {kind: "attribute",  lower:-1, upper: 1, type: "ValueType",
            desc: "Maximum value used for monitoring purposes"}

    rate:   {kind: "attribute",  lower: 1, upper: 1, type: "Integer",
            desc: "Sampling rate, if zero, only on change"}

    storage: {kind: "attribute", lower: 1, upper: 1, type: "Integer",
            desc: "Decimation factor, if zero, no storage, if > 1 all samples are stored"}

```

Figure 3-1. Component and Monitor meta class textual specification excerpt. The Component class extends the SCI (Software Configuration Item) class that defines project management features like WBS center id. Monitors, like Alarms, Commands and Control Input and Outputs are Port specializations. Descriptions are abbreviated in this example.

### 3.2 Device control modeling

Device Control Subsystems (e.g. primary mirror control system, adaptive secondary control system, etc.) command mechanical degrees of freedom or read from optical and other sensors. Degrees of freedom are represented by a set of state variables that define the observed and desired state of the system and are often updated in an arrangement of nested control loops. Although the primary function of a Device Control Subsystem is the control of hardware devices, an efficient and robust operation requires the development of:

- Calibration software that is required to produce error maps or models that allow the control system to achieve the required performance;
- Diagnosis software that provides the capability to verify the behavior of some, quite complex, hardware subsystems, especially during the commissioning phase;
- Safety functions that prevent the engagement of the interlock and safety system, which acts usually in a more ‘dramatic way’; and
- Supervision software that coordinates several components or subsystems to guarantee their correct states and that implements fault management strategies that enable the graceful degradation of performance in non-nominal scenarios.

The basic entities used to model the device control domain are Device *Controllers* and *Supervisors*. Device controllers are specialized components that implement the control function of single degree of freedom (e.g. linear position controller) or multiple degrees of freedom that coordinate more elementary ones (e.g. axis group controller). Device Controllers that must perform critical control functions are specified using Sequence Based Specification (SBS)<sup>16</sup> black-box method, that provides the basis for a Model Based Testing implementation with the support of tools like *realsbs*<sup>17</sup>, *protoseq*<sup>18</sup> or *JUMBL*<sup>19</sup>. Section 3.5 includes an example of development workflow for the specification of a Device Controller. Supervisors implement the high-level interfaces of DCSs and are responsible of the subsystem integrity (e.g. collision avoidance), component configuration, subsystem robustness, component life cycle and subsystem modal transitions amongst other functions.

Figure 2-2 provides an example excerpt of the Controller meta class that defines how to specify Controller components. The interface of Controller components, i.e. its inputs and outputs, is specified using specialized Ports that define the stimuli (i.e., commands, configuration changes, control data inputs) and the response (i.e., control data outputs, telemetry streams, alarm notifications and logs). The Motion Controllers’ stimuli response and behavior are based on the IEC

61800-7 standard<sup>20</sup>. This enables the implementation of common motion control problems (e.g. linear motion with single or dual feedback) in a straightforward way. In addition to the behavior defined by the standard, specialized behaviors are captured by the equivalent Mealy state machine result of the SBS analysis.

Section 3 describes how the information captured in the model can be used to (a) estimate global telemetry performance requirements; (b) generate test cases covering the transitions of the Mealy state machine; or (c) generate platform specific code skeletons.

```
Controller    "BasePositionController"
  extends: ["BaseController"]
  data_outputs:
    position_actual_value:
      desc:    "This output shall provide the actual value of the position
                measurement device. The value shall be given in user-defined
                position units."
      type:    "float32"
      pattern: "pull"
      max_rate: 20
      default: 0
    position_internal_demand:
      desc:    "This output shall indicate the internal position demand
                calculated by the internal trajectory generator that the drive
                should move to in position profile mode using the
                current settings of motion control properties such as velocity,
                acceleration, deceleration, motion profile type etc.
                The value of this input shall be interpreted as absolute
                or relative depending on the state of the property abs/rel'.
                It shall be given in user-defined position units."
      type:    "float32"
      pattern: "pull"
      max_rate: 20
      default: 0
```

Figure 3-2. BasePositionController specification excerpt

### 3.3 Data processing modeling

Data processing refers generally to a system that reads input data, performs some operations (analysis, reduction, conversion, etc.) and then outputs other useful or desired information. The types of processing usually involve any combination of the following operations: conversion, transformation, analysis, aggregation, sorting and validation. During operation it is necessary to process science and wavefront data generated by detectors (e.g. science, guiding, phasing, and wavefront detectors) as well as telemetry data streams, to determine associations between reference and data files, and to invoke appropriate calibration processes so as to produce GMT data products.

In order to capture the commonality between all the various data processing needs, the pipeline design pattern<sup>21</sup> is used. A *Pipeline* consists of a series of *Filter* components (modular processing elements) connected in an arrangement (series, parallel or a combination of both), where the outputs of one Filter connect to the inputs of other(s). Pipelines include also *Recipes*, which are specialized ways of connecting the pipeline filters. For each recipe configuration, parameters and top-level inputs and outputs are defined as well. Pipelines include recipes for quick-look, calibration and end-to-end data reduction.

This modular design allows the filters to be reused between different recipes, and between different pipelines. Pipelines implement a high-level interface that enables them to integrate with the rest of the observatory software, so they can be launched automatically when input data become available (event-driven). The interface also enables standardizing how and where data products are stored. Section 3.8 shows an example of how to generate a Python adapter for a pipeline.

The meta model provides a way to define data product standards using keyword dictionaries (Figure 3-3). In the example Figure 3-3, a keyword dictionary maps the FITS standard keyword (“DATAMAX”) with project-specific keywords (“datamax”), and enables the capability to identify those keywords that are metadata. Keywords can have tags associated with them. Tags allow to view or query the entire keyword set (potentially thousands) according to relevant use cases or

natural categories to which the keywords belong. This design provides the basis for the Data Archive modeling and implementation. Keyword dictionaries can obviously be used for other situations besides FITS data products.

```

KeywordDictionary "FITSDictionary",
  desc:           "Keywords for a basic FITS standard header. In many cases,
                  the keywords and descriptions are taken verbatim from
                  'Definition of the Flexible Image Transport System (FITS),
                  Version 3.0'."
  extends:        []
  max_length:     8
  keywords:
    datamax:
      tags:        ["fits_standard", "gen_primary", "data_value"]
      keyword:     "DATAMAX"
      synopsis:    "Maximum data value"
      index:       "F"
      default:     ""
      type:        "Float"
      hdu:         "any"
      desc:        "The value field shall always contain a floating point
                  number, regardless of the value of BITPIX. This number
                  shall give the maximum valid physical value represented
                  by the array, exclusive of any special values."

```

Figure 3-3. FITSDictionary specification

### 3.4 Observatory Operations modeling

To take full advantage of the outstanding image quality provided by the telescope operation modes, and to optimize for observing efficiency, several tools will aid observation. These tools are used to (a) design and plan observations; (b) estimate exposure times; and (3) operate the telescope during observing. For example, the design tool serves several purposes:

- To plan precise placements of science apertures, e.g. for creating mosaic observations, placing slit masks or fiber locations to avoid spectral overlaps
- To locate stars for telescope guiding or for AO to achieve Strehl requirements for science, factoring in target airmass, distance and brightness of AO stars relative to science target
- For non-sidereal targets, to optimize observing times and rotator orientation that maximize target and guide star tracking window

The tool components are modeled as a set of user interaction Panels organized in Screens, and a set of pipelines that estimate observing parameters based on basic observing input data. These tools generate an observation description (Figure 2-5) that is ready to be executed by the observatory Sequencer. The language that describes an observation shares the same concrete syntax as the other DSLs. It also captures and communicates the intent of the observation to both the astronomers and the observatory staff. The description of the observation is purely declarative, leaving to the sequencer the actual translation of the observer intent into telescope control components operations requests. This mapping is implemented by observing sequence templates. A library of these templates, in combination with the activation of different component connector maps, makes it possible to implement different system operations modes and configurations.

```

GMTIFS_Observation "30DORfield",
  notes:           ""
  instrument:      "gmtifs"
  obs_cond:
    image_quality: "20%/Best"
    cloud_cover:   "20%/Best"
    wind_velocity: "20%/Best"
    sky_brightness: ""
  inst_conf:

```



```

inst_rot_mode:      "tracking"
inst_PA:           135
dichroic:         "grating"      #same as grating
image_filter_wheel: "J"
image_utility_wheel: "clear"
imager_det_focus: "filter"      #same as filter
ifs_mask_wheel:    "f-converter" #same as f-converter
ifs_f_converter_wheel: "50"
ifs_filter_wheel:  "grating"     #same as grating
ifs_grating_wheel: "hK"
ifs_detector_focus: "grating"    #same as grating
sets:
  30_DOR:
    desc: "30_DOR Observation Set"
    blocks:
      pre_std_obsq:
        desc: "Pre Standard Star Observation Sequence"
        ao_conf:
          mode: "LTAO+OIWFS"
          rate: "1000"
        targets:
          base: {ra: "05:38:44.86", dec: "-69:05:44.2"}
          science: {ra: "05:38:44.86", dec: "-69:05:44.2"}
          agwfs3: {ra: "05:38:43.40", dec: "-68:59:66.8"}
          pcwfs : {ra: "05:39:25.75", dec: "-69:11:36.0"}
          oiwfs : {ra: "05:38:48.48", dec: "-69:05:32.6"}
        acquisition:
          exp_time: 5.0
          coadds: 1
          mode: "bright_object_spectroscopy"
          pause: true # Center target in IFS field
        exposure:
          det_conf:
            exp_time: 20.0
            coadds: 10
            mode: "bright_object_spectroscopy"
          offsets: [
            { p: 0.0, q: 1.1, repeat: 1 }
            { p: 0.0, q: -1.1, repeat: 1 }
            { p: 10.0, q: 0.0, repeat: 1 }
            { p: -10.0, q: 0.0, repeat: 1 }
            { p: 0.0, q: 1.1, repeat: 1 }
            { p: 0.0, q: -1.1, repeat: 1 }
          ]

```

Figure 3-4. Observing block for the GMTIFS instrument specification

### 3.5 Work system modeling

A set of modeling and development *Workflows* provides guidance to elaborate the technical specifications and the implementation of software subsystems in a way that conforms to the reference architecture. For each main class of model component (e.g. Subsystem, Controller, Pipeline) a process workflow provides guidelines for its specification, development, testing and integration. A family of workflow-centric development and modeling tools apply lean practices that automate repetitive tasks and implicitly enforces the reference architecture and standard practices. As other elements of the model, workflows are stored under configuration control to ensure the coherence between development tools and developed artifacts across different released versions. Embodying workflows into model files provides an objective foundation to implement a continuous improvement process in which A3 reports<sup>22</sup> can be associated with workflow tasks.

```

Workflow          "comp_ctrl_dev_wkf",
info:             "Controller Development Workflow"
desc:             "Control subsystem development.
                  A command line option is available to invoke each of these tasks"
roles:            ["subsystem programmer", "subsystem engineer"]
target:           "Controller"

tasks:

```

```

init:
  info: "Initialize environment"
  desc: "Initializes the development environment
        Generates the scaffolding to support the development of the subsystem"
  action: (subsystem) -> subsystem.gen_scaffold()
  tasks:
    stakeholder_input:
      info: "capture stakeholder input"
      desc: "Capture stakeholder input as requirements in natural language.
            The included action generates a requirement definition skeleton"
      action: -> @meta.Requirement.gen_instance()
    boundaries:
      info: "identify boundaries"
      desc: "identify the boundaries of the component (equivalent to identifying actors
            in use case analysis). For each device #{@meta.model.Device.info} identify
            the corresponding ports.
            The action generates the skeleton of a Controller specification"
      action: -> @meta.Controller.gen_instance()
    stimuli:
      info: "identify component stimuli"
      desc: "Component stimuli represent the input to the system and are classified in
            the following categories:
            - commands: #{@meta.model.Command.desc}
            - configurations changes: #{@meta.model.Property.desc}
            - control data inputs: #{@meta.model.Port.desc}"
    response:
      info: "identify component response"
      desc: "Component response represents the outputs of the system
            and are classified in the following categories:
            - control outputs: #{@meta.model.Port.desc}
            - alarms: #{@meta.model.Alarm.desc}
            - telemetry monitors: #{@meta.model.Monitor.desc}
            - log messages: #{@meta.model.Log_Msg.desc}"

```

Figure 3-5. Excerpt from the Controller development workflow specification.

### 3.6 Document generation

The documentation plugin automates the generation of documents that are necessary to support system and subsystem reviews and runtime operation. The plugin uses template-based generation strategy. It combines *RST*<sup>23</sup> templates with model elements to produce a set of *RST* files. These files are processed using the *Sphinx* package<sup>24</sup> that can produce documentation in *LaTeX*, *PDF* and *HTML* format. A set of predefined reports such as Design Requirement Documents (DRD), System Design Documents (SDD) or Interface Control Documents (ICD) enable the support of stage-gate documentation needs of the project. Given that more than 50 software subsystems have been identified, the capability to generate all these documents in a batch process provides obvious gains in productivity and consistency. Moreover any changes in the model will be automatically propagated to all documents in a consistent way. This also facilitates the review process as a new snapshot of updated documents can be produced in few minutes. This documentation is also available as references in the runtime system. The following lines show an example of how to generate documents in an interactive session.

```

> gmt.model.m1_cs.gen_doc "swc_drd_rpt" # generates a requirements definition report
> gmt.model.m1_cs.gen_docs()           # generates all the project documents related to the
                                       # m1 controls system (DRD, SDD, ICD, SIP)

```

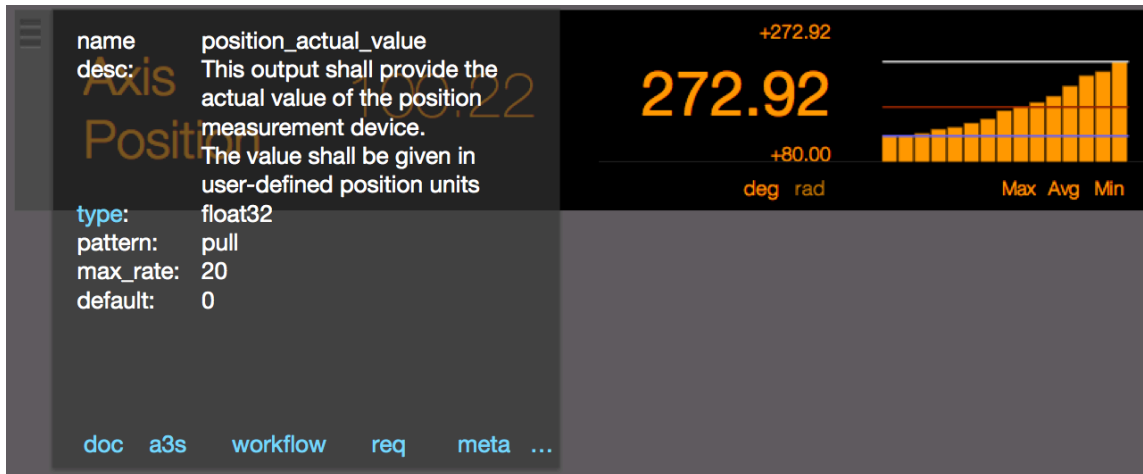


Figure 3-6. Use of meta data in the runtime. An engineering panel generated user interface HTML5 component shows how magnitude limits and unit values are extracted from the model information. An optional dialog shows access to all the model information, including traceability to requirements, extended documentation, workflow used to generate the element, and A3 reports associated with continuous improvement of this element through the life of the component. This information allows operators to get access to the information that describes the intent of the design. It also facilitates requirement traceability during testing.

### 3.7 Design statistics and metrics

As described in Section 2, the software necessary to operate and control an observatory can be modeled and explained using a set of essential concepts that span several domains of knowledge, e.g. observatory operation, device control, data processing, process sequencing, hardware I/O. The results from the domain engineering process have helped to identify the categories and number of components in each subsystem. Where a large number of components of the same category have to be developed, a framework provides a platform-specific implementation of a design architecture and a set of common base components that define the framework API. The statistics plugin generates metrics based on the model definition. It also allows the estimation of network and storage resources based on component features (e.g. telemetry monitors, control inputs and outputs). These statistics are updated regularly as the model is refined.

Table 3 shows the distribution of Component classes found from the domain analysis process. The first column of the table names the most common components. The second column identifies the number of different classes found for each component. The classes define the number of different behaviors and interface specifications. The third column lists the number of individual component instances. The number of instances defines the number of different configurations for each class; it enables one to estimate the amount of telemetry data generated and gives a measurement of the effort necessary for assembly, integration and testing, as each instance has to go through those processes individually.

Table 3. Number of Component model classes and instances.

Component	Classes	Instances
Controller	225	665
Pipeline	133	139
Server	75	75
Panel	298	302

### 3.8 Code and test generation

The code generation plugin generates skeletons of model element instances. Code generation works in a consistent way independently of the modeling layer, i.e. when producing instances of a meta model it generates the skeletons of model elements, but when producing instances of model elements it generates the skeletons of source code. When possible, rules and conventions are automated to provide consistency through the project. For example, the code generation system has knowledge of where, within a file structure, the skeleton code shall be created. This facilitates the development of building and testing tools. The model layer generator can generate adapters, implementation and configuration files, command line and visual test clients, field bus descriptions and OPC-UA specification files. The code generation plugin supports several target specific platforms. The following lines show a few examples of how to generate different types of code.

```
> gmt.metamodel.Controller.gen_instance "ml_support_ctrl" # generates the skeleton to specify a
# Controller named "ml_support_ctrl"

> gmt.model.ml_cs.gen_instance() # Generates code for all the components
# of the ml control system

> gmt.model.gen_instance "cpp", "adapter", "ml_support_ctrl"
> gmt.model.gen_instance "cpp", "configuration", "ml_support_ctrl"
> gmt.model.gen_instance "py", "adapter", "ngws_cal_pipeline"
```

The test generation plugin generates test from Mealy machines specification. It also generates test runners and validators. During testing implementations are checked against tests generated automatically from the specification. SDFs provide the information necessary to generate automated tests and test runners, which are the bases for continuous testing. SDFs are used as the base for acceptance testing of any software component.

### 3.9 Agile process support

The integration of the software components into the overall GMT system engineering effort follows tailored-compliance with the ISO 12207 standard, whereas the software development is based on an iterative Agile development process. One of the useful aspects about Agile planning is that it is based on implementing features that will be needed in the system instead of defining the tasks necessary to produce them. The key to iterative development is to frequently produce working versions of the final system that have a subset of the required features. They are fully integrated and as carefully tested before final delivery.

A major release cycle of one year is expected. Each release has a main theme and delivers a set of coherent system functions. The sum of these functions embodies the support for system level operation capabilities (e.g. support for natural seeing observing mode). Components are allocated to a target release and are assigned to lead developer. This provides the overarching framework for detailed iteration planning, and provides the necessary hooks to integrate Agile within the overall project stage-gate plan.

The planning plugin allows the creation of product backlogs or to generate reports like Cumulative Flow Diagrams, that provide insight not only on the amount of work pending, in-progress or finished, but also on the speed of team. It also enables the team to identify potential bottlenecks so it can assess how to tackle them. The plugin enables one to estimate the software costs based on the detailed description of the deliverable artifacts and on the measured cost per unit. These costs are updated after every release through the life of the project.

## 4. CONCLUSION

In this paper we have describe how the standardization of a reference architecture and a development process is implemented by means of a set of workflows and tools that efficiently generate well-formed systems that can be tested. We have described also how the use of domain specific languages and the development of a set of tools that manipulates them (a) facilitates the capture of the semantic model of the system; (b) drive the generation of interfaces, system configuration files, on-line documentation and code skeletons that integrate application code with specialized domain frameworks; (c) provide support for capturing requirements, generating test data and automated tests that ensure traceability and validation of specifications; and (d) support the definition of an end-to-end component-based system

architecture (from embedded control to high level observatory operation support) that enforces a consistent breakdown of all the software subsystems into more elementary components. This strategy facilitates the use of an Agile software management process embedded in a stage-gate project, enabling the generation of project documentation, metrics for reporting development progress, and sprint backlogs based on the priority of component features.

## ACKNOWLEDGMENTS

This work has been supported by the GMTO Corporation, a non-profit organization operated on behalf of an international consortium of universities and institutions: Astronomy Australia Ltd, the Australian National University, the Carnegie Institution for Science, Harvard University, the Korea Astronomy and Space Science Institute, the Smithsonian Institution, The University of Texas at Austin, Texas A&M University, University of Arizona and University of Chicago. This work has also been supported in part by the National Science Foundation under Scientific Program No. 10 as issued for support of the Giant Segmented Mirror Telescope for the United States Astronomical Community, in accordance with Proposal No. AST-0443999 submitted by AURA.

## REFERENCES

- [1] ISO/IEC, “ISO/IEC 12207:2008 Systems and software engineering – Software life cycle processes,” (2008)
- [2] D.C. Schmidt, “Model-Driven Engineering,” IEEE Computer 39 (2), (2006)
- [3] [http://en.wikipedia.org/wiki/Model-driven\\_engineering](http://en.wikipedia.org/wiki/Model-driven_engineering), retrieved (2014)
- [4] Fowler, Martin, [Domain Specific Languages], Addison-Wesley Professional, (2010)
- [5] European Cooperation for Space Standardization, “Space engineering - Control Systems,” ECSS-E-ST-60-10C, (2013)
- [6] Autosar, <http://www.autosar.org>, retrieved (2014)
- [7] AADL, <http://www.aadl.info>, retrieved (2014)
- [8] Eclipse, <http://www.eclipse.org/modeling/emf/>, retrieved (2014)
- [9] OMG, <http://www.omg.org/mof/>, retrieved (2014)
- [10] Voelter, Markus, [DSL Engineering – Designing, Implementing and Using Domain-Specific Languages], (2013)
- [11] <http://esprima.org>, retrieved (2014)
- [12] <http://git-scm.com>, retrieved (2014)
- [13] Bernstein, R., Johns, M., McCarthy, P., Raybould, K., Bigelow, B., Bouchez, A., Filgueira, J., Jacoby, G., Sawyer, D. and Sheehan, M., “Overview and status of the Giant Magellan telescope,” Proc. SPIE 9145-47, (2014)
- [14] McLeod, B., Bouchez, A., Espeland, B., Filgueira, J., Johns, M., Norton, T., Ordway, M., Podgorski, W., Roll, J., Smith, C., “The Giant Magellan Telescope active optics system,” Proc. SPIE 9145-64, (2014)
- [15] Bouchez, A. et al. “The Giant Magellan telescope adaptive optics program,” Proc. SPIE 9148-31, (2014).
- [16] Prowell, S. et al., “Foundations of sequence-based software specification,” IEEE Transactions on Software Engineering 29, p. 417-429, (2003)
- [17] [realsbs.sourceforge.net](https://realsbs.sourceforge.net), retrieved (2014)
- [18] <http://sourceforge.net/projects/protoseq/>, retrieved (2014)
- [19] Prowell, S., “Jumbl: “A tool for model-based statistical testing,” Proc. HICSS-03, p.337.3, (2003)
- [20] IEC, “IEC 61800-7 ed1.0 Adjustable speed electrical power drive systems – Part 7: Generic interface and use of profiles for power drive systems,” (2007)
- [21] Gamma, et al., [Design Patterns: Elements of Reusable Object-Oriented Software], Addison-Wesley, p. 383, ISBN 0-201-63361-2, (1994)
- [22] Poppendieck, M. and T., [Leading Lean Software Development: Results Are not the Point], Addison-Wesley, ISBN 0321620704, (2014)
- [23] <http://en.wikipedia.org/wiki/ReStructuredText>, retrieved (2014)
- [24] Sphinx-doc.org – Python Documentation Generator, retrieved (2014)