PROCEEDINGS OF SPIE

SPIEDigitalLibrary.org/conference-proceedings-of-spie

PORIS toolkit DSL applied to instrument development and implementation

Jacinto Javier Vaz-Cedillo, José Antonio Rodríguez-Losada, Enrique Joven, Angel Manuel Bongiovanni, Alessandro Ederoclite, et al.

Jacinto Javier Vaz-Cedillo, José Antonio Rodríguez-Losada, Enrique Joven, Angel Manuel Bongiovanni, Alessandro Ederoclite, Antonio Cabrera-Lavers, Gabriel Gómez-Velarde, Roberto González-Rocha, Patricia Fernández-Izquierdo, Miguel Ángel Torres-Gil, Noé Rodríguez-González, Jorge Quintero-Nehrkorn, "PORIS toolkit DSL applied to instrument development and implementation," Proc. SPIE 12187, Modeling, Systems Engineering, and Project Management for Astronomy X, 121870B (25 August 2022); doi: 10.1117/12.2629602

SPIE.

Event: SPIE Astronomical Telescopes + Instrumentation, 2022, Montréal, Québec, Canada

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022 Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

PORIS Toolkit DSL applied to instrument development and implementation

Jacinto Javier Vaz-Cedillo*[ab], José Antonio Rodríguez-Losada[a], Enrique Joven-Álvarez[c], Ángel Manuel Bongiovanni[ef], Alessandro Ederoclite[df], Antonio Cabrera-Lavers[a], Gabriel Gómez-Velarde[a], Roberto González-Rocha[b], Patricia Fernández-Izquierdo[cb], Miguel Ángel Torres-Gil[cb], Noé Rodríguez-González[b], Jorge Quintero-Nehrkorn[cb]

[a]Gran Telescopio de Canarias, S.A. (Grantecan), E-38205, La Laguna, Tenerife, Spain

[b]Asociación Cultural para la Promoción de la Robótica y el Estudio del Cosmos (CosmoBots), E- 38004, Santa Cruz de Tenerife, Spain

[c]Instituto de Astrofísica de Canarias (IAC), E-38200, La Laguna, Tenerife, Spain

[d]Centro de Estudios de Física del Cosmos (CEFCA), E- 44001, Teruel, Spain

[e]Institut de Radioastronomie Millimétrique (IRAM), E-18012, Granada, Spain

[f]Asociación Astrofísica para la Promoción de la Investigación, Instrumentación y su Desarrollo (ASPID), E-38205, La Laguna, Tenerife, Spain

ABSTRACT

Model-based systems engineering has as one of its central pillars the single source of truth that is usually a CAD model, or a model defined using a language such as SysML. However, having a single point of truth is not incompatible with using multiple modeling languages. A simple DSL like PORIS allows us to make instrument sketches much more concise and understandable than if we made them in SysML. By providing this language with transformers, we can automatically and instantly generate configuration panels, diagrams and documentation that allow the scientific team of the instrument to create more quickly and formally the configuration and functional specifications of the instrument. Engineers can also create a high percentage of the instrument software, for instance, the ones related to configuration, monitoring, diagnostics or safety. In this article we will show how, starting from a simple model in a spreadsheet, we will end integrating its software in the GTC control system.

Keywords: MBSE, DSL, PORIS, GCS, GTC, Configuration, Safety, GUI

1. INTRODUCTION

In model-based systems engineering, models themselves take a central role, as they are the basis for launching a large fraction of essential project processes.

A single model is not enough to represent all the aspects of a system that are relevant during its development, so in many cases it is necessary to develop several models. Being able to keep the different models in sync is desirable and convenient.

Modeling languages such as UML or SysML have been developed to allow engineers to model almost all kinds of problems or solutions. The complexity required to give SysML such versatility creates an opportunity for simpler, more intuitive languages to enter the picture and play partial but key roles in the project.

If they are designed to be simple and easy to understand for people without training in system modeling, they allow you to keep the team connected to the model, like how a storyboard acts during the production of a movie.

Models created with this simpler language can be synchronized with SysML models to avoid feature overlaps. This allows the team to be connected with the models developed in SysML through this simpler language.

In this article, we use the PORIS language, described in document at Ref. 1, as a proof of concept to explore the desirability of simpler models. Figure 1 shows some highlights of the language.

Modeling, Systems Engineering, and Project Management for Astronomy X, edited by George Z. Angeli, Philippe Dierickx, Proc. of SPIE Vol. 12187, 121870B
© 2022 SPIE · 0277-786X
· doi: 10.1117/12.2629602

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

PORIS is much simpler than SysML, because it reduces the scope of the aspects of the system that can be modeled with it. It is much less powerful, its models can be enriched with much fewer attributes, many types of relationships that cannot be modeled with PORIS. But on the other hand, its simplicity makes it easy for domain experts to understand, thus allowing them to be part of the modeling team, which helps prevent the project from falling into many technical pitfalls.

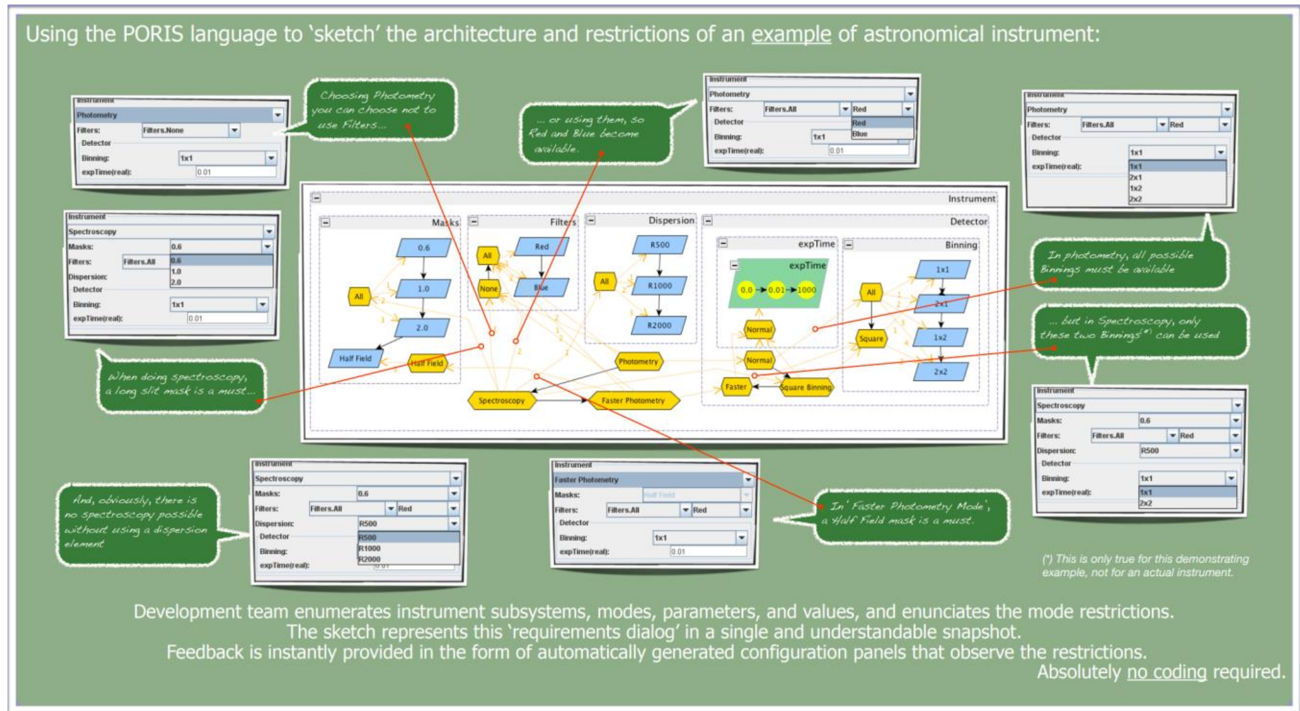


Figure 1. A fragment of the poster related to the PORIS article in Ref. 1, issued in 2010.

This article shows some real experiences of modeling with PORIS and obtaining valuable products thanks to the PORIS toolkit, including the automatic production of documentation, diagrams and software.

As a proof of concept, the goal of this article is not to propose the PORIS language or its set of tools, but rather to encourage other projects to model certain aspects of their system with simpler and more convenient languages.

PORIS was used for developing the OSIRIS MaskDesigner tool, described at the GTC website[1] and in the article at Ref. 2 and the article in Ref. 6. OSIRIS instrument is described in the article at Ref. 3 and at the GTC website[2].

2. MODELING THE CONFIGURABILITY OF A SYSTEM WITH PORIS

As discussed above, the simplest languages perforce ignore most aspects of the system. Therefore, to produce value, the few aspects of the system to be modeled must be chosen wisely, so that the benefits obtained far outweigh the efforts invested not only in modeling with it, but also in building synchronization mechanisms with the SysML models.

PORIS was designed to represent the configurability of a system. We consider configurability to be a small but extremely important aspect of understanding a system, especially in fields like astrophysical instruments, which involve many parameters and many diverse operating modes.

[1] http://www.gtc.iac.es/instruments/osiris/osirisMOS.php

[2] http://www.gtc.iac.es/instruments/osiris/osiris.php

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

It is also a difficult aspect to keep under control of the human development team, since although it is easy to define what parameters, each subsystem must have and what ranges of values are valid for each parameter, it is difficult to define how the value of a parameter in a specific context may be conditioning the applicability or range of valid values for another parameter.

Furthermore, it is quite common to develop not a single system, but a product line. In those cases, product variants can be modeled as different modes of operation for the same system, whereby a specific product is implemented as a system in which higher-level configuration decisions have been made at design time.

Keeping track of product variants is difficult from a development team member's point of view, because documentation becomes confusing, and design and implementation become complicated. It becomes difficult and time consuming to anticipate how specification changes will affect all product variants. Using the same language to model product variants and configurations minimizes team effort and allows the team to account for variants. An example of this will be later shown in Figure 30, where the different firmware variants a controller can run are modeled as values of a system parameter being defined.

2.1 An example of the PORIS model of an instrument

To gain some familiarity with reading diagrams in PORIS language, we present a model of a simple astrophysical instrument in Figure 2.

This instrument is capable is capable to acquire images in three possible modes: Photometry, FastPhotometry, and Spectroscopy.

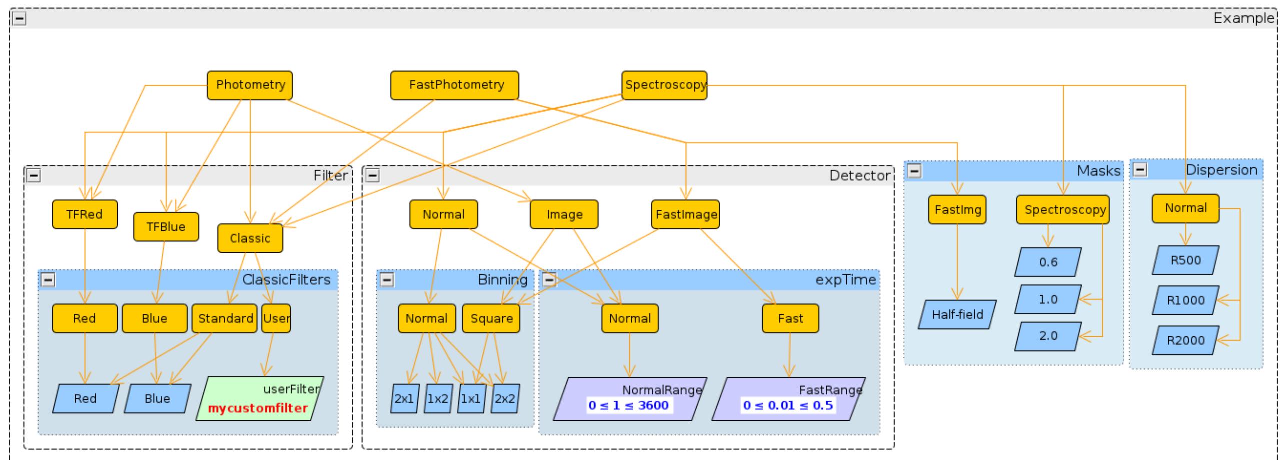


Figure 2. An example of an instrument modeled in PORIS.

Figure 3 shows some screenshots of an interactive instrument configuration panel. These configuration panels are automatically created and executed by the PORIS toolkit from the diagram. Automatic generation of this effective feedback artifact takes less than 20 seconds and involves no human intervention.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

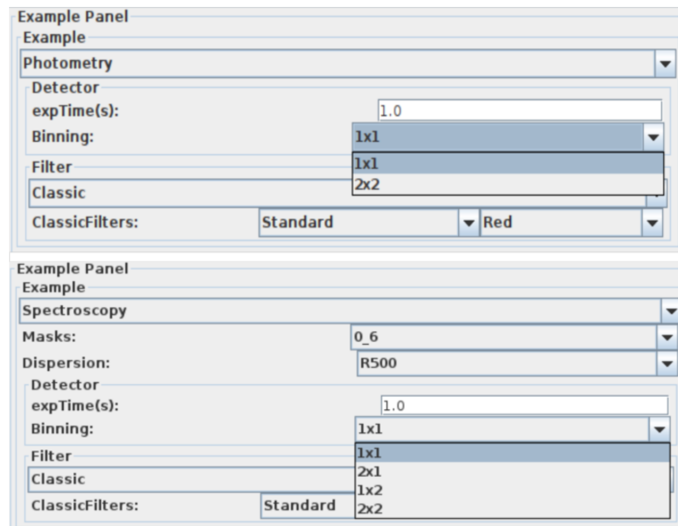


Figure 3: Two screenshots of the configuration panel automatically generated from the example model shown at the previous figure. In the upper one, Photometry operating mode is selected, and the user is selecting the detector binning, allowing only 1x1 and 2x2 values, corresponding to the Square binning mode. In the lower one, the Spectroscopy mode is selected, and the user is selecting the detector binning from all possible values.

By looking at the diagram with the model (which is supposed to be agreed with all domain experts) and using the configuration panels, any member of the team can have at a very early stage, a very precise idea of the capabilities of the future instrument and will be able to determine if a configuration is valid or aberrant.

Communication supported by executable panels are less sensitive to bad communication. As stated in the article at Ref. 5 some communication problems as “The Curse of Knowledge” can be occurring when the scientists (experts) are transmitting the knowledge to the engineers. The instant feedback loop makes these undesirable situations occur less and be shorter. In a document it is likely to occur that two people are reading different information from the same sentence. Having two different views of the same model (diagrams and panels), helps clarifying the obscure concepts.

Even without prior specific training, a newcomer to the team can infer how the PORIS syntax represents the configurability of the system through the simple and intuitive juxtaposition of the diagram and the observed behavior of the panel. After a few iterations, the newcomer should be able to incorporate changes into the model and advance their modeling skills by seeing how the changes alter the panel.

2.2 Modeling a system from scratch

Let's say we want to model an astronomical instrument that contains a camera. In this example situation, our domain experts (the astronomer, the organization's detector expert), told the modeler that the camera can operate in two modes: normal or fast.

Fast mode uses a half-field mask, so only half of the detector is exposed to light during the exposure time. The use of the mask allows to move the charge acquired by the detector from the illuminated side to the dark side, allowing the detector to continue acquiring photons while the electronic converters read the charge of the dark zone of the detector. This technique allows obtaining images at a higher speed. On the contrary, it divides the field obtaining images of half size. Exposure time is another parameter that the user should be able to set.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

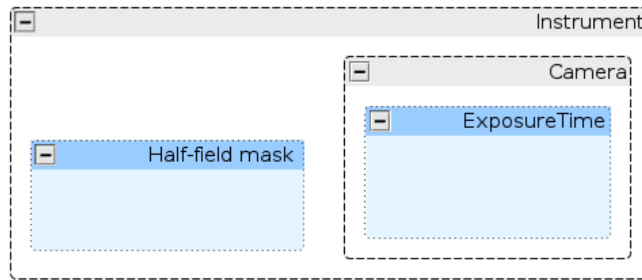


Figure 4: First modeling step, simply adding the systems and parameters to establish a hierarchical architecture.

The systems engineer in charge of modeling the instrument divides the information received by domain experts into two main sets:

- Information that can be modeled with PORIS, because it falls within the scope of the language.
- Information that, while important, cannot be modeled with PORIS and must be preserved for incorporation into models written in SysML or text-based specification documents. This information cannot be modeled as it falls outside the instrument configurability aspect covered by PORIS.

The information about the movements of the electric charge, the techniques to accelerate the frequency of image acquisition, the technical justifications behind the rapid acquisition strategy that domain experts are transmitting to us belong to the set of information that cannot be transcribed into language PORIS.

The components of the system, their parameters, the values that those parameters can take, the modes of operation of the components, and the relationships between these modes of operation and the values of the parameters can be modeled using PORIS1.

The modeler identifies the 'Instrument' system, and the 'Camera' subsystem, and an instrument-level parameter called 'Half-field Mask'. Figure 4 describes the result of this modeling step.

The diagram is drawn using the yEd graph editor, and it is stored in a file named `instrument.graphml` inside the example's directory of the project.

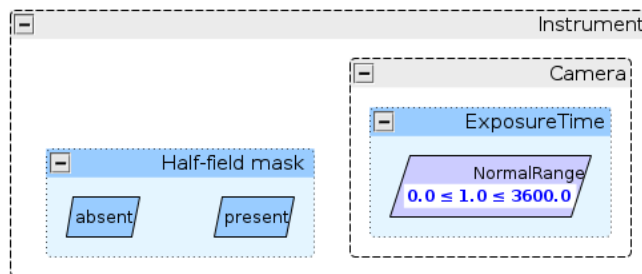


Figure 5: Second modeling step, adding possible values for the parameters

An important parameter of the camera is the exposure time that we define as a floating-point value that can range between 0 and 3600 seconds, values that were given by domain experts after consulting them. Another relevant parameter is the presence or absence of the half-field mask. Let's define the values that this parameter can take as 'absent' and 'present'. Figure 5 is showing the model at this step. Note that he PORIS syntax forces the modeler to choose an intermediate number

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

inside the valid range of the ExposureTime parameter. The expression '0.0 ≤ 1.0 ≤ 3600.0' describes the valid range of the Exposure time [0.0,3600.0] and the default value of the parameter (1.0).

It seems easy to foresee that setting the camera's detector to fast mode will somehow cause the half-field mask to be 'present', while 'absent' is the correct value when setting the camera to 'normal' mode.

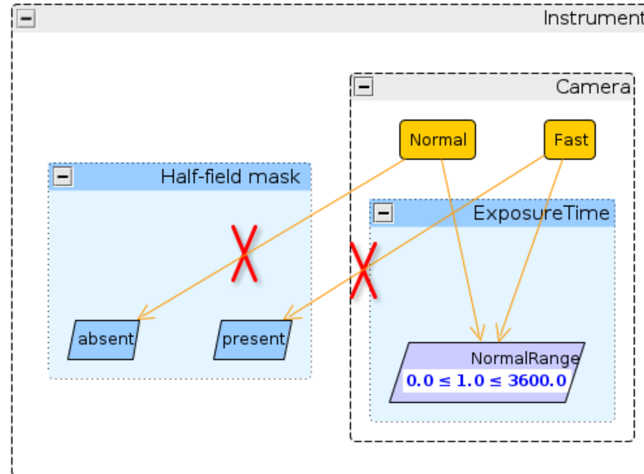


Figure 6: Intuitive model (but syntactically wrong) that includes the operating modes of the instrument. Here the camera modes affect the behavior of a sibling component. The PORIS language states that modes can only affect the behaviors of child components.

PORIS has been designed to keep all its relationships hierarchical, avoiding interference between components that are not part of a parent-child relationship. Each component offers its handling interface to its parent component, and only to it. The handling interface is restricted to being a list of operating modes of the child that can only be selected by the parent's operating modes.

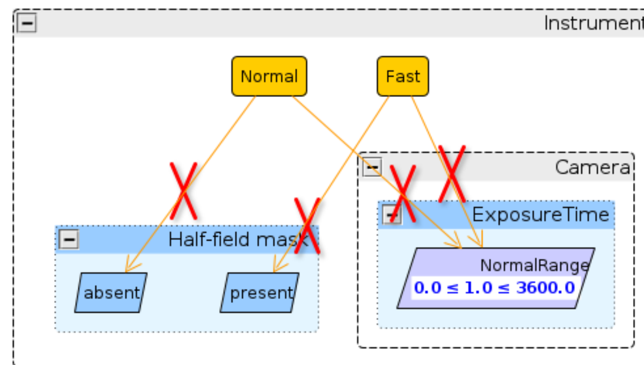


Figure 7: Intuitive (but also syntactically wrong) second model of the instrument. Instrument modes affect ExposureTime, a parameter that is not in a child element, but in the child element of a child element. In the case of the half-field mask parameter, the instrument modes directly affect its values. In PORIS, modes can only affect the values of the component itself or the modes of child components.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

Figure 6 shows the intuitive representation of the behavior described for the instrument. However, we can see that the camera modes impact the values of a sibling parameter. The diagram violates the language syntax: the half-field mask handling interface provides direct values (not modes) to the modes of a sibling (not a parent) component.

Figure 7 partially resolves the syntax issues: here the camera modes have been elevated to the instrument level, so that there is no relationship between sibling elements. Unfortunately, the number of levels the relationship is traversing (2 instead of one in the case of the ExposureTime parameter) and the target type of these relationships (values instead of modes) also violate PORIS syntax.

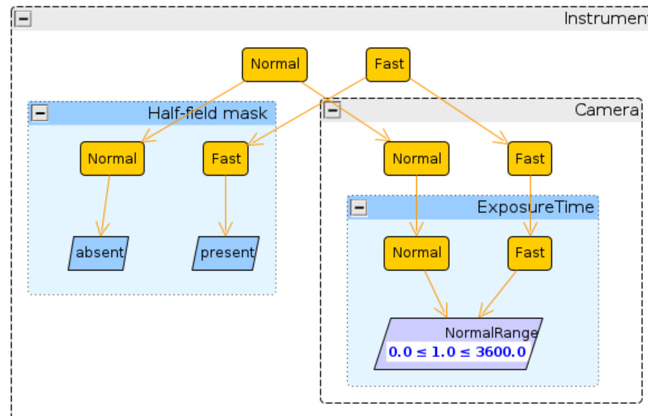


Figure 8: Instrument model rewritten to comply with PORIS syntax.

Figure 8 shows a model of the instrument that is correct in PORIS language. We have seen that the original relationship that linked the modes of the camera with the modes of a mask has required some not entirely intuitive transformations:

- The camera mode choice has been elevated to an instrument mode, so that all parameters that should be affected by it become part of the model sub-tree under that choice.
- To determine which value is selectable in each parameter depending on the parent component's modes of operation, a handling interface has been added. For example, to allow selecting 'absent' or 'present' values for the half-field mask, 'normal' and 'fast' modes have been created at the same parameter level.
- Since relationships cannot pass through multiple levels, intermediate modes have been created to allow instrument level modes to reach original destination modes. For example, to force selection of the 'present' value of the half-field mask parameter when the user selects 'Fast' as the Instrument's operating mode, the 'Fast' instrument mode is propagating the selection to the 'Fast' half-field mask, which is causing the 'present' value of parameter to be selected.

Once a first valid PORIS model of our instrument is achieved, the PORIS toolkit can be used to analyze this diagram and create an instrument configuration panel to explore our instrument's configurability with. This is done in a single sentence, and it takes less than two seconds to create the panel. The statement to get the panel is as simple as:

```
./porispanel.sh example/instrument
```


Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

Figure 9 shows the behavior of the instrument configuration panel, automatically created from the model. This panel is interactive and allows the user to analyze the configurability of the model. The panel is a valuable artifact that allows the domain expert to check how the supplied specification is understood at the systems engineering level.

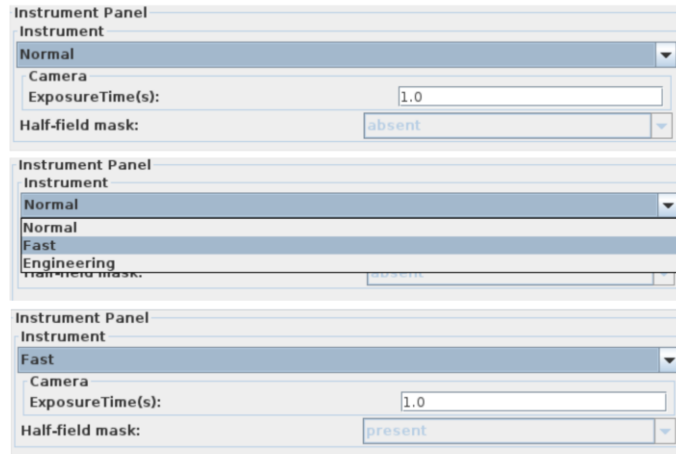


Figure 9: Configuration panel showing the behavior of the instrument, automatically generated from the model.

The figure shows also the effect of changing the instrument's 'Normal' operating mode to 'Fast' operating mode, causing the value of the half-field mask parameter to change from 'absent' to 'present'.

It is important to point out that the user of the configuration panel will never be able to directly choose the value of the half-field mask parameter, since it will always be automatically calculated based on the choice of the instrument's operating mode. If the user tries to set a value for the exposure time outside the range [0.0,3600.0], the configuration panel will prevent it.

The model (through the interactive panel) is guiding the user in the configuration of the instrument, offering only the applicable values of the applicable parameters, and even checking that the data entered conforms to the applicable ranges for it. In this way you are preventing the instrument from being configured in an aberrant way.

The panel is ready to be sent to domain experts for double feedback:

- To verify that what is specified by the experts is being understood and applied correctly from the point of view of the systems engineer,
- To check that the specification delivered to the systems engineers was the one that the experts were willing to communicate.

2.2.1 Model refinement

After receiving feedback from the expert, it is quite common to need to refine the model. As the experts have had an artifact to play with, some spec issues have surfaced, which were harder to see. In this way they agree that part of the specification was missing, and they agree to correct this lack.

In the present case, the refinement is justified according to an expansion of the knowledge about how the detector works in its fast mode, which makes us realize that there are exposure time settings that do not make sense. Thanks to this refinement, we will be able to reduce the space of valid configurations, correctly qualifying as aberrant all those configurations for which the experts, now, do not find sense.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

By iterating with the panel, the experts were able to verify that the user could have exposure times of 3600 seconds in the fast mode of the instrument. Experts immediately realize that this setup doesn't make sense, why try to go faster by sacrificing half the detector if the exposure time is much longer than the detector's read time? Experts understand the need to improve the specification so that the possible value for exposure time in fast mode is more restrictive than for normal mode.

But, not least, they understand that the panel is behaving as they specified, because they understood and agreed with the model diagram from which the panel was generated without human intermediation. They cannot, therefore, confuse the specification problem with a panel implementation error.

This type of situation, repeated over time, makes experts pay more and more attention to the diagram, since they understand that the diagram is not only descriptive, but also executive. And they recognize it as a vehicle that transports their influence directly, without intermediaries, from the specification to the behavior of the system. Experts therefore feel empowered and will most likely increase their level of commitment to the modeling activity.

Once the need for refinement is agreed upon, comes the next series of questions: what should be the restrictive range for the exposure time in fast mode? Is it convenient to also allow a very low exposure time?

The experts of our example are of the opinion that, in fast mode, there is only one recommended value for the exposure time since, to optimize said mode, the detector must always be exposing in its light part and reading in its dark part. The value of the exposure time that achieves this result is the one that equals the reading time of the darkened half of the detector.

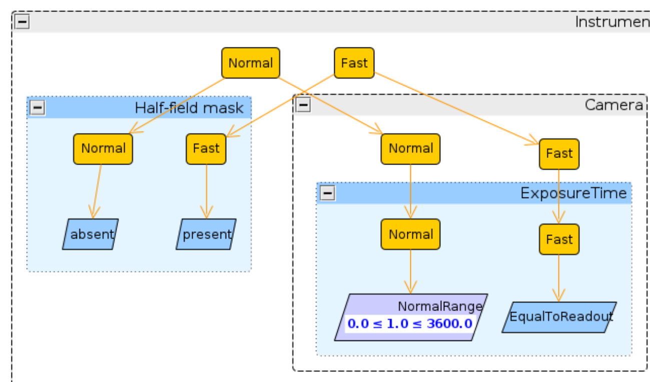


Figure 10: Including the EqualToReadout value for the ExposureTime parameter in the model.

After a productive team discussion, the experts agreed that when fast detector mode is active, the only valid option for exposure time should be a new value called 'EqualToReadout'.

This value is not specified with a number, it is left as a label, thus leaving it to the engineers to calculate it in the design phase. That number will most likely depend on the detector and control electronics that will eventually be integrated into the final instrument, so its calculation at the specification stage is not only not necessary but also not possible.

Figure 10 shows how the model has changed to reflect this refinement. Comparing this diagram with the previous one, it can be agreed that their introduction could have taken a few minutes. Seconds after the refinement was introduced, the experts in our example received the configuration panels shown in Figure 11.

The evolution that the diagram will undergo after the following refinement will be easy for the reader to foresee once we state it: for calibration purposes, experts agree that a Bias mode is needed to obtain a complete image of the detector after an exposure time of 0.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

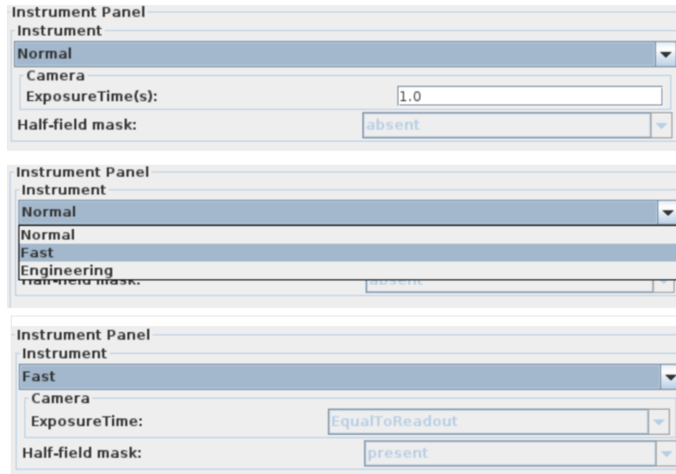


Figure 11: Panels after adding the EqualToReadout value to the model.

This could be achieved with the detector in normal mode by selecting an exposure time equal to zero, but the astronomers in our example argue that it is very convenient to have a specific observing mode so that the images can be labeled as 'Bias'. That way, those images will be easily identifiable for inclusion as calibration data within scientific reduction recipes.

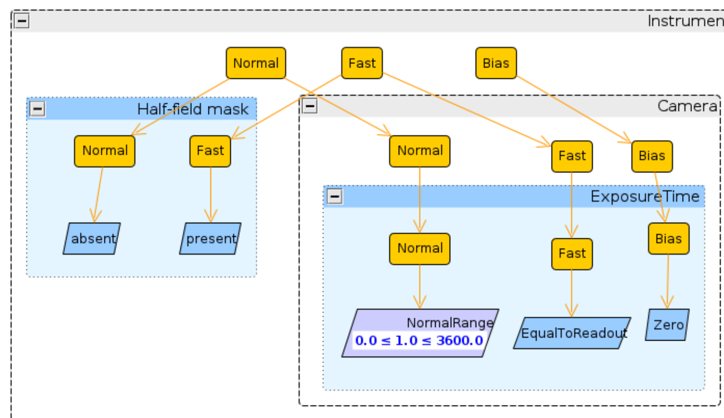


Figure 12: Bias mode introduced in the PORIS model diagram

Since our imaginary team is experimenting with an instant model of the instrument from the very moment it is being developed, it can talk from the beginning about aspects of the instrument that, such as the management of the metadata that will accompany the images, are usually addressed later.

Figure 12 shows the model after adding the Bias mode, Figure 13 shows the panels corresponding to the change. The midfield mask parameter is not relevant for Bias mode, since the detector does not acquire light, so Bias mode has nothing to do with the midfield mask.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

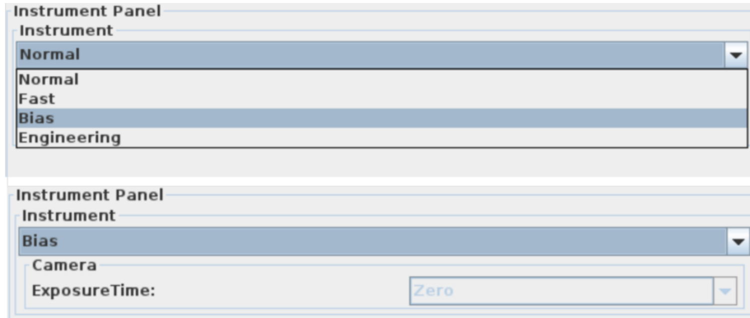


Figure 13: Instrument panel after adding the Bias mode.

Because feedback is so immediate the modeler can be assisted by experts while they are in the meeting, minimizing interruptions to specification activity. The specification activity, compared to that only supported by the documentation, not only flows with fewer interruptions and requires less effort, but the percentage of the modeling time in which the modeler benefits of the direct support of the experts is higher. In addition, most of the review and refinement of specifications tangent to the instrument configuration tends to be done earlier in the project schedule.

2.3 Some notes about PORIS

After this practical introduction on the benefits of using the PORIS language and its way of doing it, we will present some information on both.

2.3.1 PORIS language

The PORIS language was described in the article at Ref. 1.

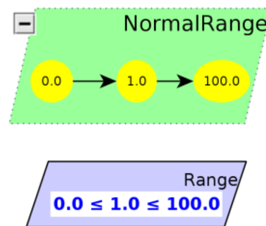


Figure 14: In 2021, the PORIS syntax for a floating-point range changed to make it easier to parse the diagrams. From the representation above, which consumes up to 6 widgets (with 4 text labels), to the one below, which only uses one widget with two labels.

During the last years some small modifications have been made to the language, without affecting its concepts, strategy or scope. The modifications are just pragmatic adaptations made to facilitate the parsing of PORIS diagrams, thus facilitating the development of new toolkit features.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

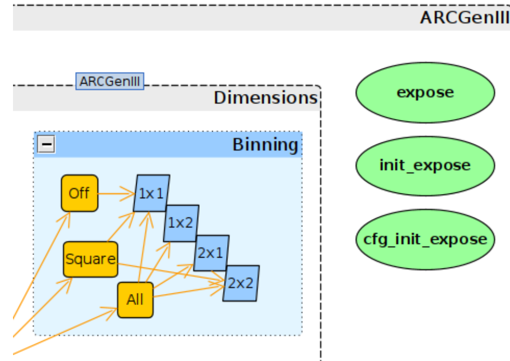


Figure 15: The green ovals in this diagram define three events that operate at the level of the subsystem that contains them (named ARCGenIII in this case).

An additional symbol was added to allow the modeler to define events that act at the level of a specific subsystem of the PORIS model. An example of these triggers is shown in Figure 15. The use of these triggers will be shown later in this article.

2.3.2 PORIS toolkit

The toolkit is a set of software tools that, in combination with other pre-existing tools, allows obtaining valuable products directly from a PORIS model.

The toolkit is in continuous improvement, and is currently composed of:

- A palette for the yEd[3] editor, so that the user can use it to draw the PORIS diagrams and store them as GraphML[4] files.
- A parsing tool that processes yEd diagrams (single diagram or a folder of diagrams), to extract the model.
- A generator that exports the model to a spreadsheet (LibreOffice[5] ODS file).
- A generator that exports the model to a PORIS-XML representation.
- A Java Swing[6] panel engine capable of reading, at runtime, PORIS-XML files to create configuration panels for the system described in the model.
- A generator that converts the model to a Python class.
- A generator that converts the model to a C++ class.
- A cosmoSys plugin that transforms a cosmoSys-Req instance into a PORIS model database and editor. cosmoSys is described at document Ref. 4. This allows, among other features:
    - Model import/export using enhanced ODS files, including conditional formatting, syntax highlighting, offline model editing aids, DSM (Design Structure Matrix[7]), etc.

---

[3] https://www.yworks.com/products/yed
[4] https://en.wikipedia.org/wiki/GraphML
[5] https://www.libreoffice.org/
[6] https://en.wikipedia.org/wiki/Swing_(Java)
[7] https://dsmweb.org/ describes the Design Structure Matrix (DSM) tool.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

    - Generation of reports from LibreOffice templates using the Carbone.io language.
    - Automatic generation of reference documentation.
    - Tree editors, to easily organize model nodes.
    - Collaborative modeling environment for teams whose members perform certain roles, including functions such as wiki, task management, discussion forums, etc. Easy to adapt so that modeling activities fit the organization's processes.
    - Hierarchy and dependency diagrams, navigable and available in web or SVG format.
    - Traceability between GraphML diagrams drawn in yEd and cosmoSys model database. Users can jump from nodes in the GraphML diagram to the corresponding inputs in the web application with a single keypress on their keyboards.
- A generator that converts a PORIS model into a complete GCS device, including:
    - All instrument configuration software:
        - At control room: instrument configuration panels.
        - Integrated in the instrument hardware: configuration management software.
    - All communication components that support the configuration and event triggering of the instrument, including the middleware entities tailored to the instrument.

The planned future features of the toolkit are:

- Bidirectional synchronization between the PORIS model and the central SysML model.
- Bidirectional synchronization between the model diagrams and the cosmoSys database.
- Automatic generation of text-based requirements from the PORIS model.
- Automatic test generation for instrument control software and instrument GUI.
- Traceability between the test results and the PORIS model.
- Automatic web-based GUI panels of the model in cosmoSys, to allow instant feedback while editing the model in cosmoSys.
- Automatic Python-based GUI panels of the model in cosmoSys.
- Automatic generation of ROS nodes from the PORIS model.
- Additional python libraries to handle PORIS devices.
- It focuses all code generation to start from the model's Python class, rather than generating from ODS or PORISXML files.
- A custom editor based on 3D views, which could allow better rendering of the PORIS model enriched with additional attributes, display links to other models, and support innovative modeling techniques that improve understanding of the modeling system and productivity.
- Etc.

The development strategy of some of the planned functions could be changed to use SysML as the source model, if an effective and efficient bidirectional synchronization between both languages is achieved. This would imply, for example,

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

that the automatic generation of text-based requirements would start from the SysML model instead of from the PORIS model.

2.4 The PORIS model of OSIRIS instrument

OSIRIS instrument PORIS model is shown in this chapter, as an actual example. We can check the OSIRIS online description4 in which the key features of the instrument are included.

Creating a document that could exhaustively describe the entire configurability of OSIRIS instrument will take dozens of pages of tables and diagrams. In PORIS language, this description can occupy a single diagram, as shown in Figure 22.

Figure 16 shows the hierarchy of these diagrams to make up the entire model.

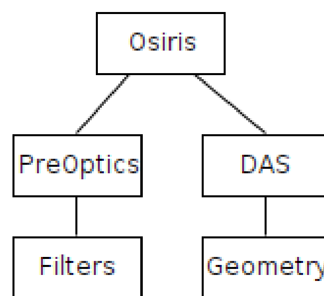


Figure 16: Diagram hierarchy of the PORIS model of the Osiris Instrument

For convenience, when the diagram representing an instrument model grows, the PORIS toolkit allows it to be divided into simpler diagrams representing parts of the same system, as we can see in figures 17 for the Filters subsystem, 18 for the OpticalPath subsystem, 19 for the DAS (Data Acquisition System) subsystem, 20 for the DAS Geometry subsystem, and 21 for the OSIRIS instrument.

OSIRIS is a versatile instrument that combines photometry and spectroscopy modes and includes special functions such as narrow band imaging thanks to its tunable filters or multi-object spectroscopy. Such a complex configurability can be comprehensively represented by four legible diagrams that can be printed on DIN A4 sheets of paper, or by a single perfectly printable diagram on a DIN A3 sheet of paper.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

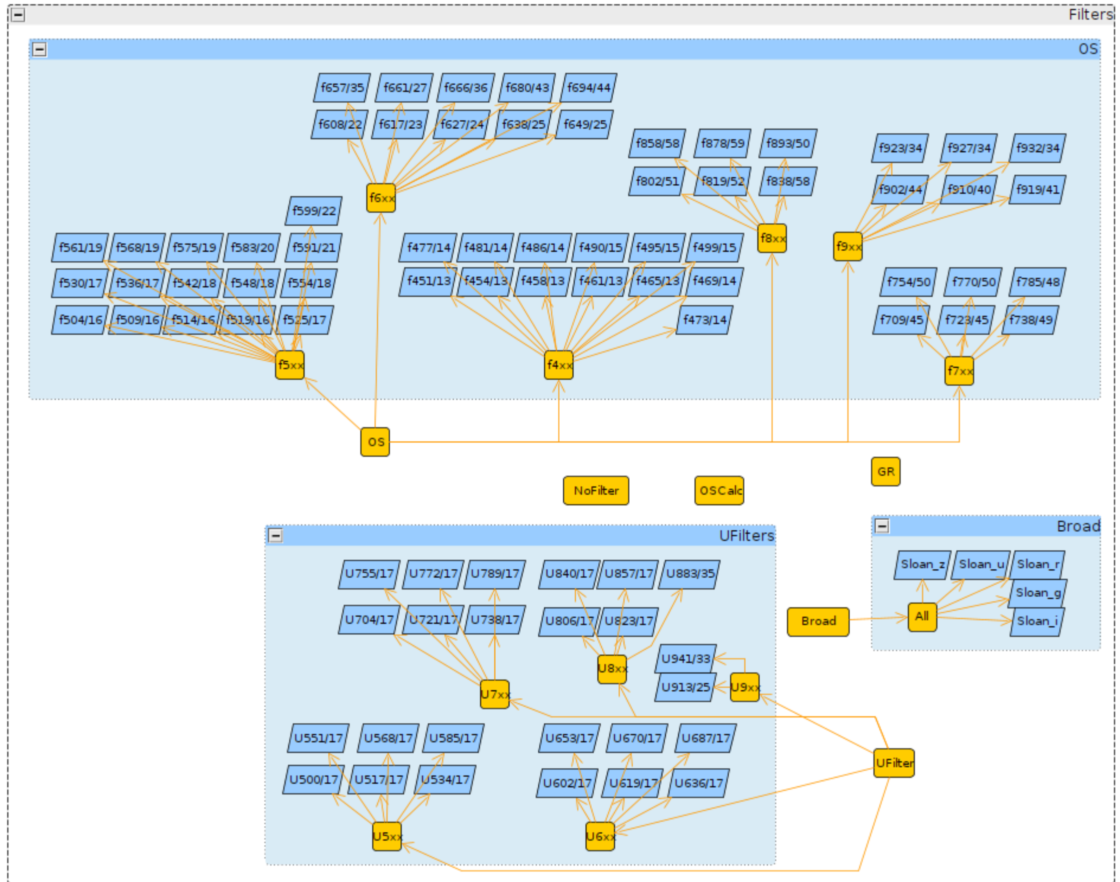


Figure 17: PORIS model of the OSIRIS pre-optical filter subsystem. The reason why some intermediate modes like f5xx or U6xx have been introduced is simply for convenience: it allows users to have shorter option lists when choosing which filter to use. The handling interface of this subsystem is made up of the NoFilter, OSCalc, GR, OS, Broad and UFilter modes.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

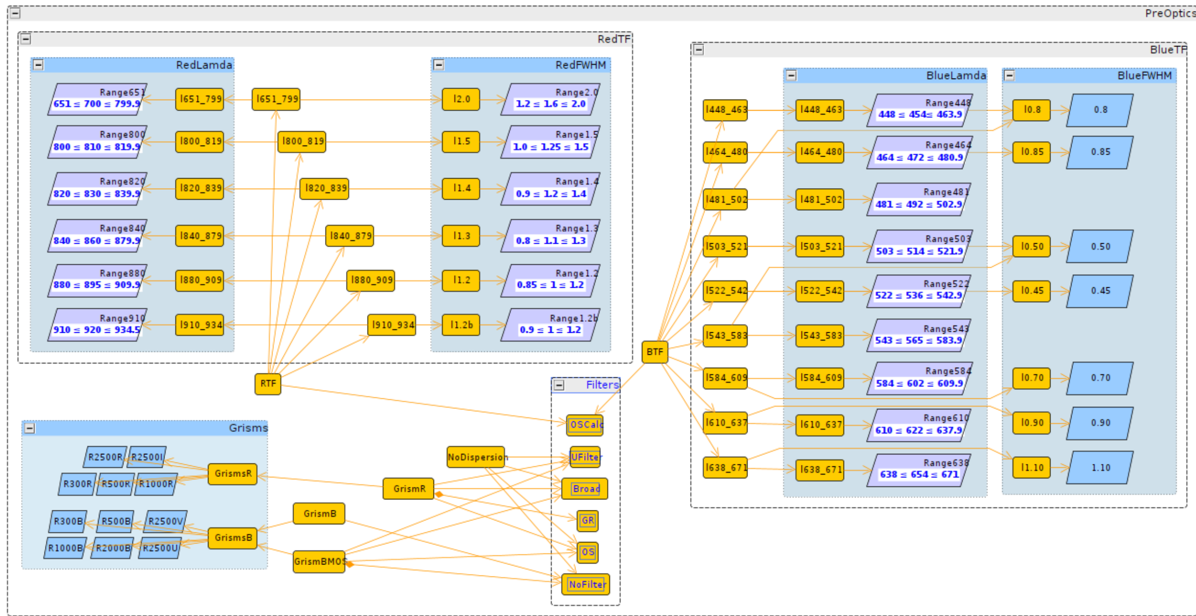


Figure 18: PORIS model of the OSIRIS PreOptics subsystem. Its handling interface is made up of the NoDispersion, GrismB, GrismBMOS, GrismR, RTF and BTF modes. It makes use of the handling interface of its Filters subsystem.

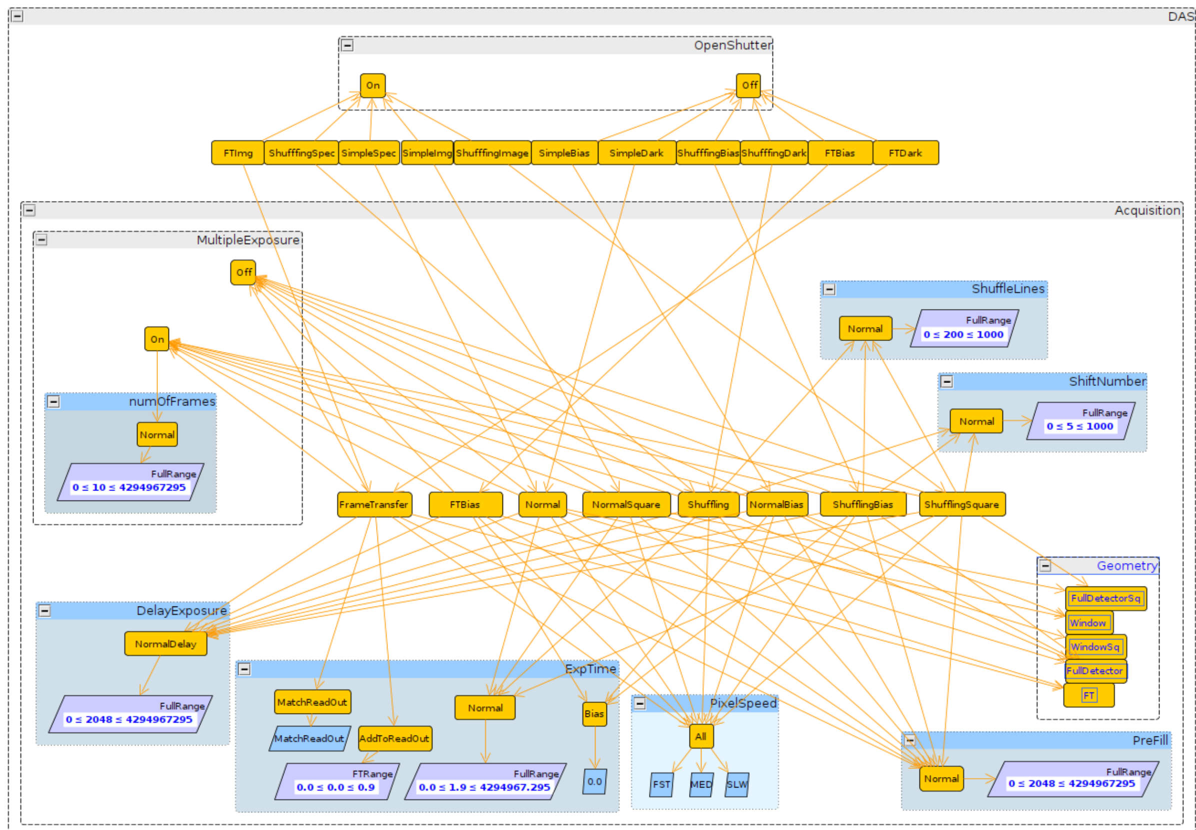


Figure 19: PORIS model of the OSIRIS Data Acquisition System (DAS). Its handling interface is made up of 11 modes shown at the top of the diagram, in a row from FTImg to FTDark. It uses the handling interface of its Geometry subsystem.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

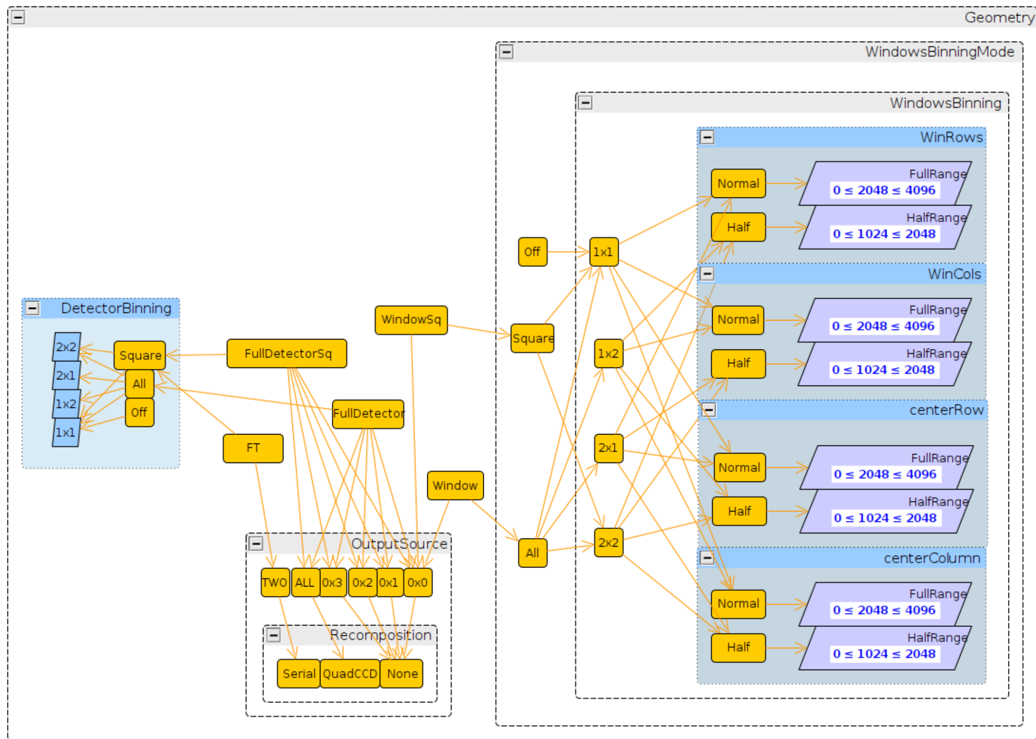


Figure 20: PORIS model of the OSIRIS DAS Geometry subsystem, which allows the instrument to use the full detector or to use a window whose valid parameter value ranges depend on the image acquisition configuration (especially the selected detector binning). This mode offers its handling interface to its parent, DAS. The interface is composed of FullDetector, FullDetectorSq, Window, WindowSq and FT modes.

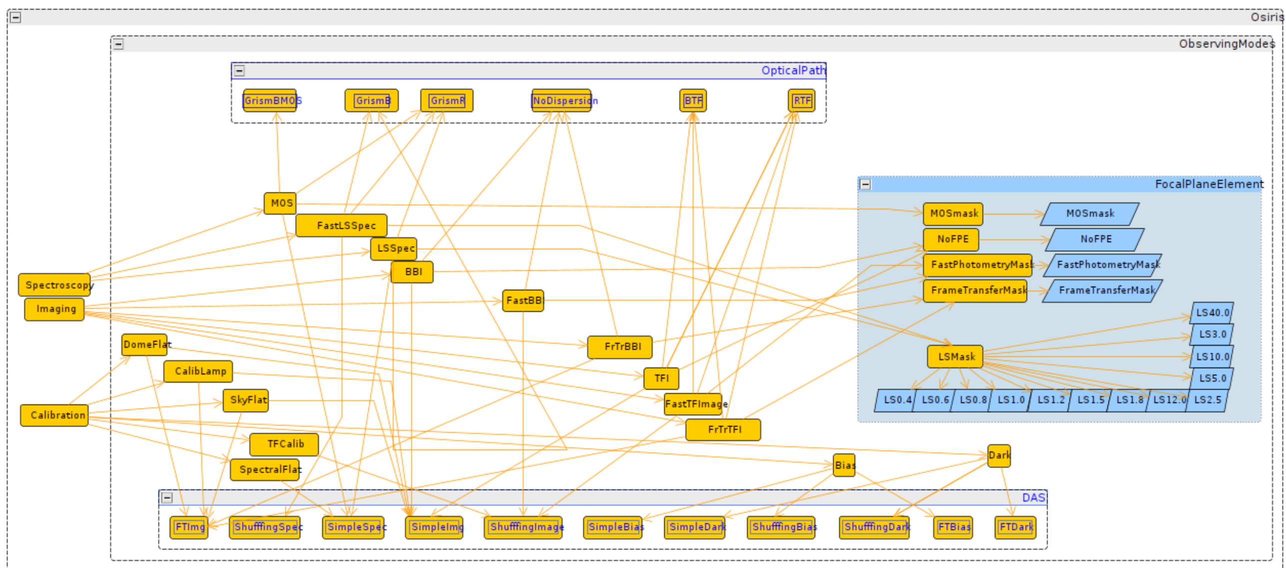


Figure 21: PORIS model of the OSIRIS instrument, which is relying in the DAS and OpticalPath subsystems, described as external to this diagram.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

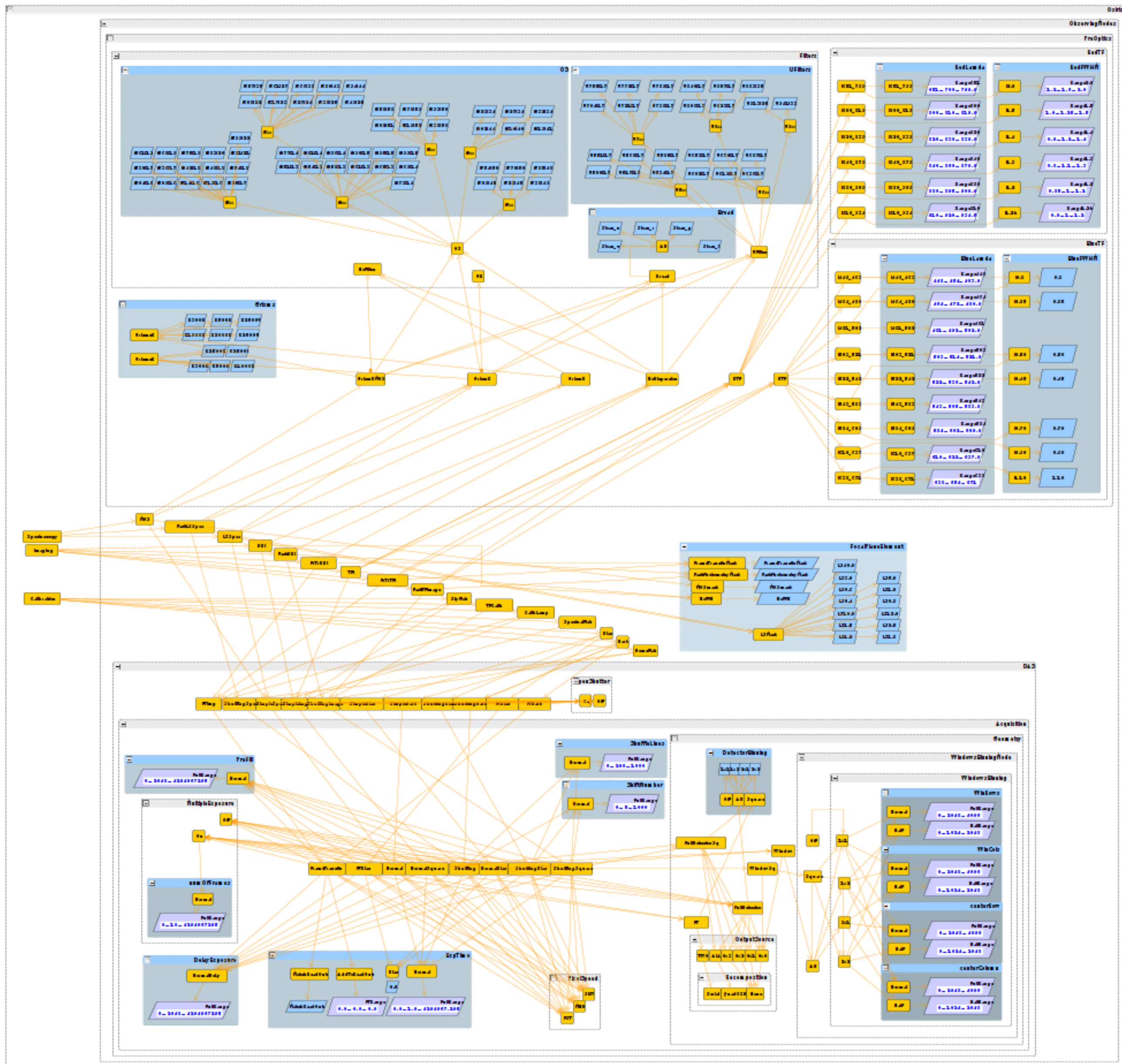


Figure 22: Monolithic diagram showing the full configurability of OSIRIS. This diagram can be easily read in a printed copy on a DIN A3 size sheet of paper.

Before the PORIS toolkit included the ability to split a model into several simpler diagrams, OSIRIS could be represented on just one sheet of DIN A3 paper. An example of this monolithic diagram is shown in Figure 22.

The files containing the modular PORIS diagrams are named `osiris.graphml`, `osifilt.graphml`, `osidas.graphml`, `osigeom.graphml` and `osiopts.graphml`. All of them are in a diagram folder called `osiris`.

To bring the configuration panels to life, we use the following command from the PORIS toolkit:

```
./porispanel_dir.sh osiris
```


Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

Figure 23 shows the configuration panels of OSIRIS instrument.

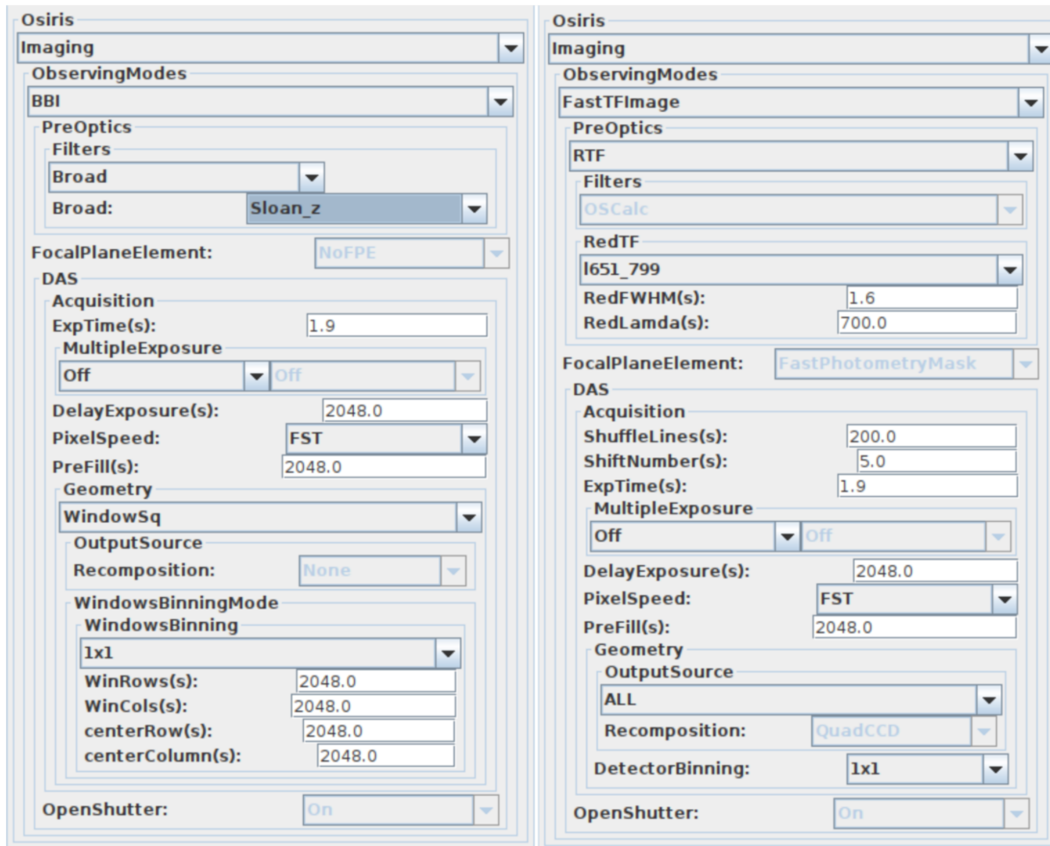


Figure 23: Configuration panels for OSIRIS instrument, automatically generated by the PORIS toolkit from the model. The one on the left shows a broadband imaging setup that allows the user to define a window, while the one on the right shows a fast tunable filter imaging setup using the full detector.

3. CREATING GCS DEVICES FROM ITS PORIS MODEL

The GCS (Grantecan Control System) is a distributed control system for telescopes, which uses middleware to distribute functions over a network of devices. These functions are grouped into devices. An astronomical instrument like OSIRIS is a device.

Devices can be virtual or physical, virtual devices are just software devices that run on computing resources (computers, embedded systems, PLCs...) while physical devices also have sensors and/or actuators.

OSIRIS is an example of a physical device. It has actuators that allow the light coming from the telescope to be processed and it has sensors to acquire data, its detector being the most important.

A physical device can be understood as a virtual device complemented by software capable of managing the hardware elements of the instrument, as illustrated in Figure 24. The physical device contains the hardware elements, managed in

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

the first instance by their drivers. To implement the instrument features that the different modes of operation will configure and activate, software engineers develop specific custom software called 'hardware driving software.'

The PORIS toolkit automatically generates the virtual-physical connector to provide a stable framework that makes the hardware control software as immutable as possible against model changes. Although the virtual device is automatically rewritten each time the instrument model changes due to refinement activity, the virtual-physical connector is intended to contain as many of those changes as possible, so that they do not affect the hardware driving software. Sometimes the virtual-physical connector will also need to be auto-generated after a model change, but the chance that there will be a need for changes to the hardware driving software becomes unlikely. Code written by engineers is therefore largely protected against model refinements occurring at the level of domain experts, allowing refinements to occur until near the end of the project without causing dramatic problems. Furthermore, this makes the instrument software highly maintainable and adaptable throughout the lifetime of the instrument.

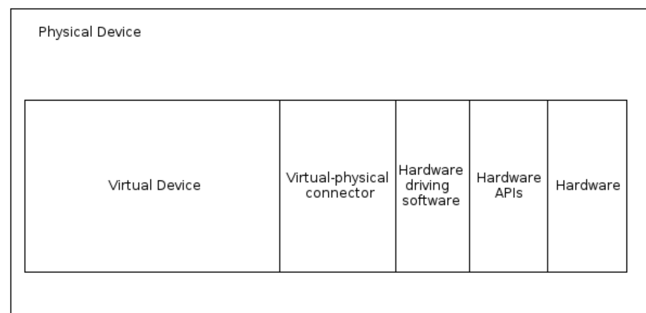


Figure 24: A convenient representation of a GCS physical device to illustrate how PORIS toolkit generates code for such devices.

The software of a physical device is deployed on the GTC network in at least two physical places:

- The LCU: The hardware APIs, hardware driving software, and virtual-physical connector run on LCUs (Local Control Units), along with part of the virtual device software. In the case of the OSIRIS Data Acquisition System, its software runs on a Linux computer (acting as the LCU) and is primarily written in the C/C++ language. This LCU is at the Cassegrain focus of the GTC telescope.
- The control room: Telescope operators can use the configuration and control panels from any computer on the GCS network capable of displaying them. For simplicity, we can say that the control and configuration panels are displayed in the control room. These panels are primarily written in Java and can be embedded as plugins to the GTC control console called Inspector. The panels for the OSIRIS instrument are part of the OSIRIS virtual device.

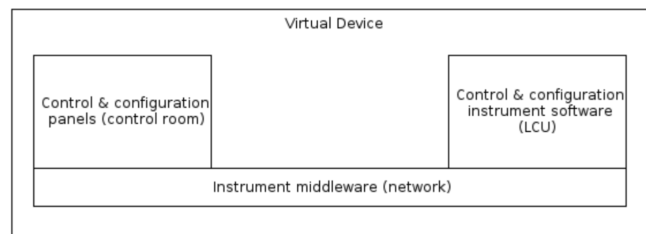


Figure 25: Software elements of a Virtual Device

Other key software that a GCS device needs is that which defines its interface in the middleware. The GCS middleware uses the CORBA standard. The middleware interface software includes definition of parameters, values, and commands in IDL files, and functions to wrap and unwrap the parameters, values, and commands to/from the middleware to/from the different software units of the device. This software is also part of the virtual device.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

The software elements that make up the virtual devices are shown in Figure 25.

A representation of how the OSIRIS physical GCS device software is deployed is shown in Figure 26

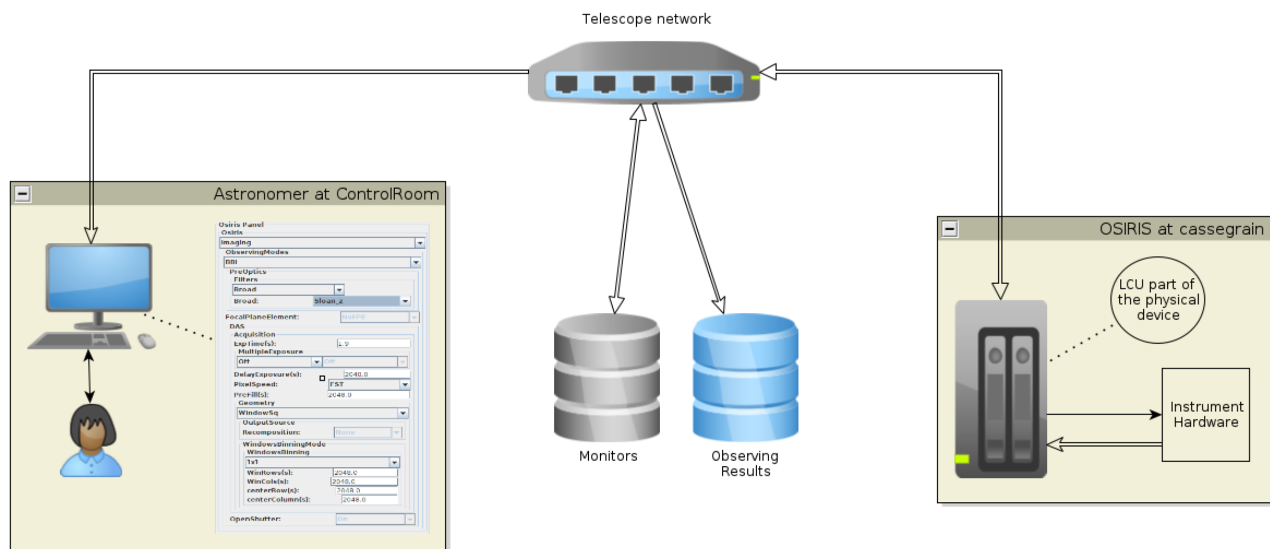


Figure 26: An example of deploying an instrument as a GCS device operating at telescope

When a user configures the instrument in the control room and presses the button to apply the configuration to the instrument, the corresponding configuration commands sequence is sent through the middleware from the control room to the LCUs. The CORBA tools guarantee that the commands implemented in C/C++ in the LCUs can be called from the control panels implemented in Java, and guarantee that there will be no problems with the conversion of the arguments or results of said commands.

Instrument status can be easily monitored using GCS monitors. These monitors are used so that any node in the GCS network can know its current status, and so that post-processes can consult its history. The PORIS toolkit automatically creates monitors for all parameters and subsystems present in the model.

Once the toolkit generates the virtual device for the first time, the user can start the instrument software on the LCU and can open the configuration panels in the control room to configure the instrument. When a configuration is applied, it is sent from the panels to the LCU, and the LCU will store it if it deems it valid for the instrument. The instrument monitors will be updated accordingly.

3.1 An example of a real physical device created using a PORIS model

To demonstrate how to use the PORIS toolkit to create a physical device, we will use a real-world example: the ARCGenIII GCS test device used in the LISA lab to control and test the new OSIRIS detector running on the GCS. The ARCGenIII GCS test device is a precursor to the Osiris DAS device currently in development. Its model is similar to the sum of the models shown in figures 19 and 20.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

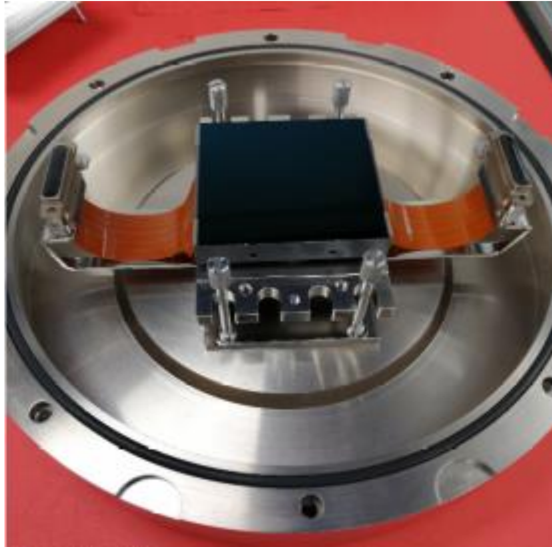


Figure 27: The new OSIRIS detector, Teledyne (e2V Technologies) CCD 231-84

The ARCGenIII GCS test device takes its name from the San Diego GenIII detector controller product family from Astronomical Research Cameras, Inc.5 The OSIRIS detector is controlled by an ARC66 PCIeX1 interface board plugged into the LCU computer. The interface board is connected by fiber optic link to an ARC-70 controller housing containing the detector control electronics, consisting of:

- ARC73 Power board
- ARC32 Clock board.
- ARC47 Four channel CCD video board.
- ARC22 Timing board

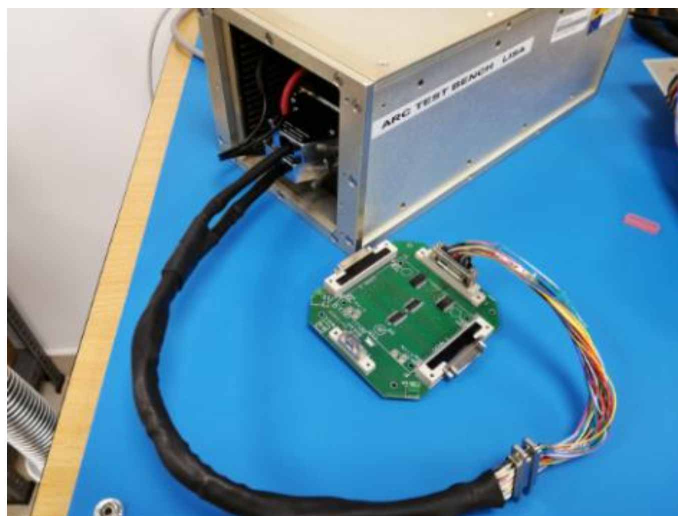


Figure 28: ARCGenIII controller, including wiring and the cryogenic interface of the detector.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

The ARC22 timing board contains the detector's control firmware, which has been modified by the experts at the IAC's LISA detector laboratory, to add some special functions required by OSIRIS.

The ARCGenIII GCS test device was modeled incrementally. These are the most notable steps:

- The first version of the model was based on the ARCGenIII programming API. The parameters were chosen from the ARCGenIII API commands and command arguments. Valid parameter ranges were simply those that the ARCGenIII API commands could accept. After building this device, we were able to validate the integration between the control room, the LCU software, and the ArcGenIII API.
- Some API commands that were not applicable to OSIRIS were removed from the model.
- Parameter values and their valid ranges have been reduced to fit OSIRIS restrictions. For example, the API allows the user to specify any positive integer for the binning factor of the image pixels, while OSIRIS restricts these values to just '1' or '2'.
- Iterating with experts in detectors from the LISA laboratory, and with astronomers from OSIRIS, the dependencies between the parameters were modelled.
- As the LISA lab experts introduced some improvements to the firmware, the commands and arguments to activate or configure these improvements were incorporated into the model.
- To support regression testing of firmware modifications, the firmware variant was entered as a device parameter. The user can choose, as part of the device configuration, between five different firmware variants to upload to the controller.
- Three command triggers have been added to the model to allow the user to command image acquisition in three different ways. Figure 29 illustrates these triggers:
    - Entire imaging pipeline activation, including hardware power-up, reboot, initialization, firmware download, and exposure loop initiation. Hardware configuration and pipeline parameters are read from the ARCGenIII device configuration model.
    - Triggering the acquisition pipeline without commanding hardware power on, reset, and firmware download. The pipeline parameters are read from the ARCGenIII device configuration model.
    - Simply launching the exposure loop, with the same parameters as the last exposure loop.
- To support the development of control software and post-processing software (image reconstruction, FITS file construction, etc.), a mode choice has been incorporated that allows the user to work in real mode (commanding the hardware) or in emulated mode (using synthetic images).

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

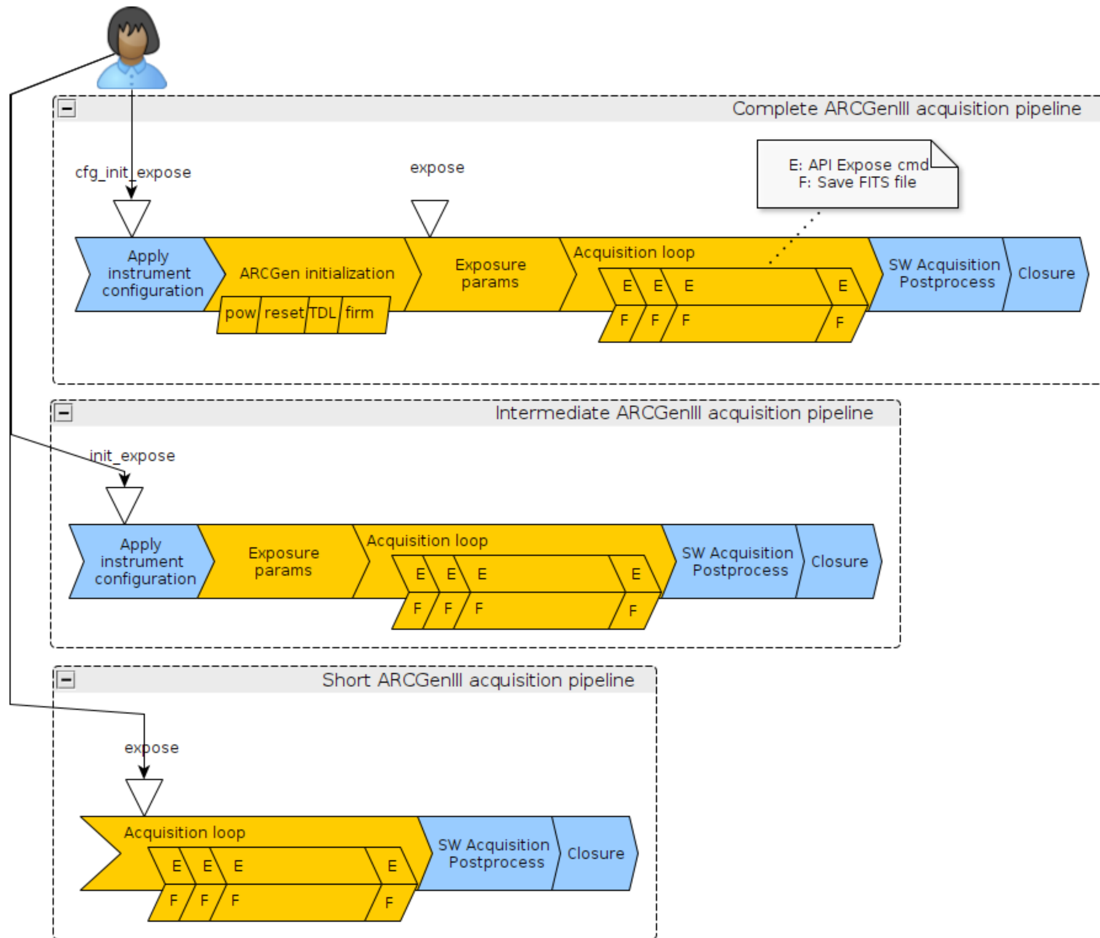


Figure 29: The three event triggers of the ARCGenIII test GCS device. The blue color denotes an activity that involves only software, while the orange denotes access to hardware.

The resulting PORIS model for the ARCGenIII test GCS device, stored in a file named `csysARC.graphml`, is shown in Figure 30.

To obtain a fully functional virtual appliance from the model, the following PORIS toolkit command was run within the GCS programming environment:

```
./PORIS/redoPorisDev.sh csysARC
```


Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

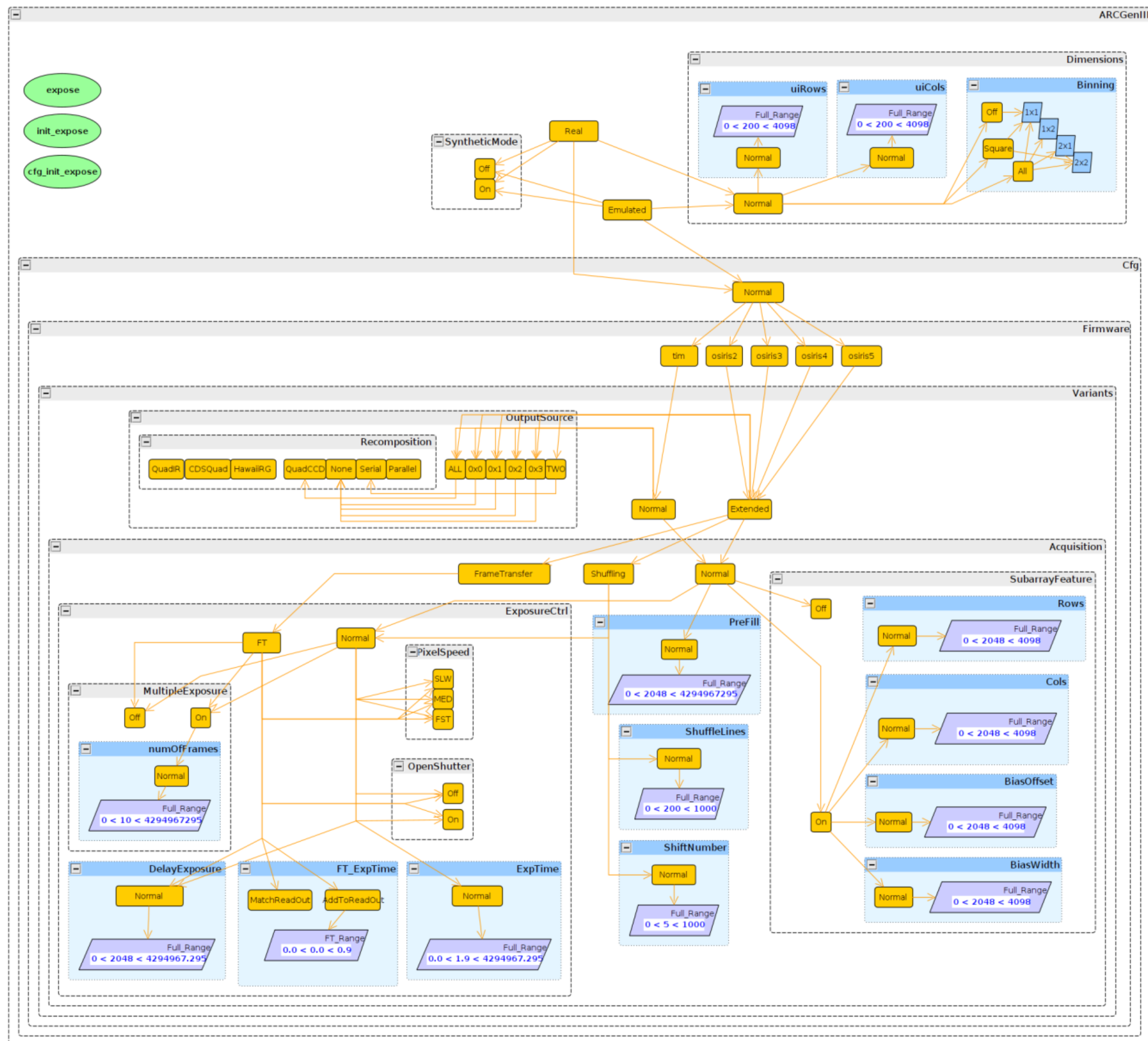


Figure 30: The PORIS model of the ARCGenIII test GCS device.

After the command completes, both the LCU software and the control panel binaries have been successfully created.

The LCU software can be started with the command:

```
csysARC/devRun.sh
```

The configuration and control panels of the device can be launched thanks to the command:

```
PORIS/deviceCtrlFrame.sh csysARC
```


Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

The configuration panels displayed in this step can be seen in Figure 31.

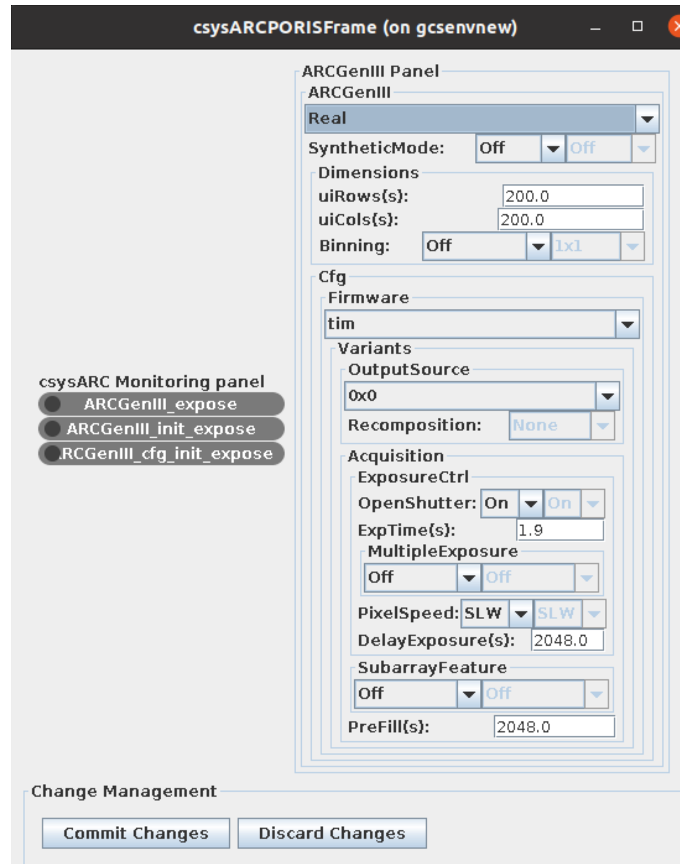


Figure 31: Configuration panels for the ARCGenIII test GCS device connected via middleware to control software running on the LCU. Since the device has just booted up, it is running but its settings are not set. Therefore, there are no monitors displayed in the left half of the control panel.

Figure 32 shows the result of applying a configuration to the instrument. After selecting the mode options and configuring the parameter values, the user must press the 'Confirm Changes' button to apply the configuration to the device. This causes configuration commands to be issued from the control room configuration panel to the control software running on the LCU. The settings are not applied to the hardware since the virtual device does not yet know how to interact with the hardware. Therefore, the configuration is kept in the memory of the device control software. Each parameter or mode is also published as a GCS monitor variable. The control room dashboard is subscribed to all those monitoring variables, colloquially called 'monitors'. As the monitors take on a known value, they appear in the left half of the control panel.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

Figure 32: Configuration panel after the user configures the instrument in emulated mode using the 'osiris2' firmware variant and setting the OutputSource to 'ALL'. In the left half of the panel, in blue capsules, the device monitors have appeared shortly after the user has pressed the 'Commit Changes' button.

The configuration panel also creates an action button for each event trigger in the model. When pressed, the LCU software receives the corresponding command invocation. The automatically created event handling function simply prints a message to the log file.

At this point we have seen what the PORIS toolkit can do fully automatically from the model. As mentioned above, the PORIS toolkit also creates some support files that facilitate the integration of external libraries, which make up the so-called virtual-physical connector. In this example, the PORIS toolkit created a folder called `csysARC.user` where, in addition to the virtual-physical connector, a mock-up of the event handling functions is implemented. This set of mock-up functions acts as a starting point for the development of the hardware driving software.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

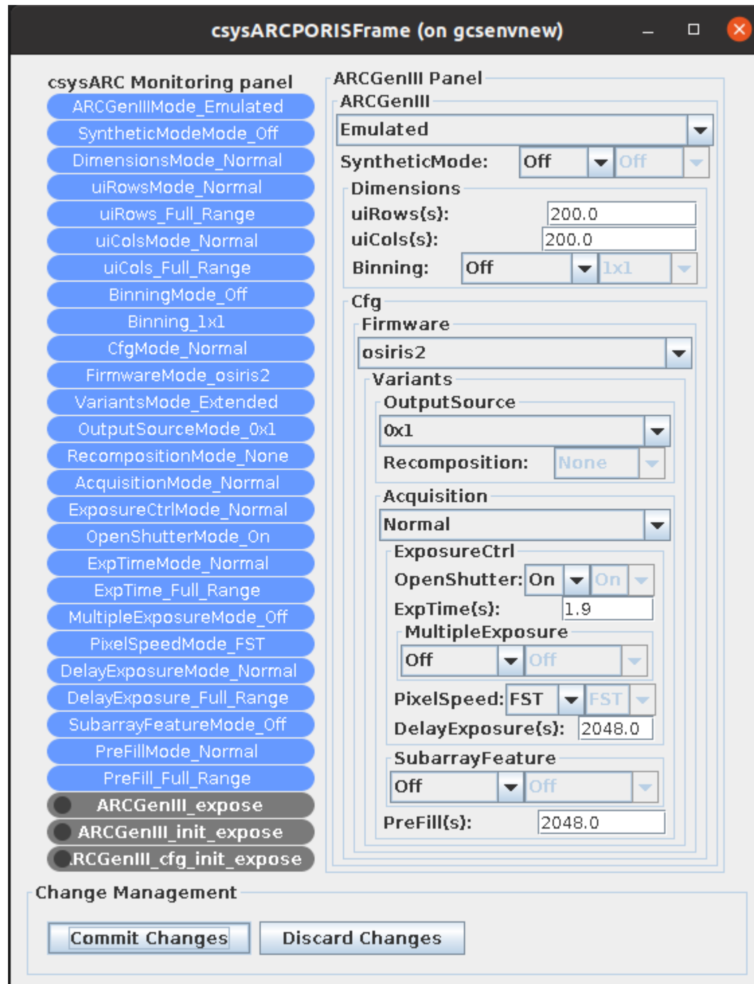


Figure 33: Configuration panel after changing the OutputSource parameter to 0x1 and the PixelSpeed parameter to FST.

The strategy followed to maximize the immutability of the interface between the virtual and physical parts of a device consists in using the object that represents the PORIS model of the instrument as the only argument of the event management functions. Thanks to this model, which can be interrogated for the current configuration, the functions can obtain the values of the parameters or modes that they need. The way in which the functions interrogate the model to get the parameters is also independent of the architecture of the model itself.

This is part of the automatically generated code for the 'Commit changes' event of the configuration panels:

```
...
public boolean commitChanges() {
...

csysARC_.setOpenShutterMode(CSYSARC.OpenShutterMode.from_int(OpenShutter.getModes().indexOf(OpenShutterCfg.get
Mode())+1));
...
```

This is the automatically generated LCU C++ code that receives the mode change request for the OpenShutter subsystem:

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

```
csysARC::OpenShutterMode csysARC::setOpenShutterMode(OpenShutterMode newmode)
{
...

  if ( newmode != enumOpenShutterMode_){
    result = (OpenShutterMode)prOpenShutter.setMode((uint8_t)value);
      if (result != enumOpenShutterMode_){
        sync_PORIS_Model();
        logInfo_('setOpenShutterMode() completed successfully');
      } else {
        logInfo_('setOpenShutterMode() not completed, the value is not eligible?');
      }
  } else {
    result = newmode;
    logInfo_('setOpenShutterMode() the mode was already set');
  }
```

The function `prOpenShutter.setMode()` tries to set the mode 'newmode' as the current mode of the OpenShutter subsystem. If this change is going to make the current device configuration aberrant, `setMode()` will prevent it, rejecting the change. If the change is accepted, `setMode()` will recursively propagate the change to lower levels of the model, following the dependency relationships. If the current value of a parameter becomes unacceptable, a default value compatible with the current mode will be set. After `setMode()` completes, a valid device configuration is guaranteed.

The `sync_PORIS_Model()` function simply copies the changes that have occurred in the instrument configuration to the instrument's GCS monitors.

At this point we have seen how the configuration flows from the panels to the LCU. We will now introduce how event trigger functions are invoked from the panel for the instrument to execute on its LCU.

This is the auto-generated code for the ARCGenIII test GCS device 'cfg init expos' event within the Java GUI code:

```
csysARC_.execARCGenIII_cfg_init_expose(true);
```

The `execARCGenIII_expose()` command is carried through middleware from the control room to the LCU. This is the automatically generated code that receives the command on the LCU side:

```
bool csysARC::execARCGenIII_cfg_init_expose(bool value)
{
...

  bool result = ARCGenIII_cfg_init_expose(&prARCGenIII);
  if ( result == EXIT_SUCCESS) {
    logInfo_('execARCGenIII_cfg_init_expose() completed successfully');
  } else {
    logInfo_('execARCGenIII_cfg_init_expose() NOT completed');
  }

...
  return(result);
}
```

As we can see, the `ARCGenIII_cfg_init_expose()` function call is using a reference to `prARCGenIII`, which is an object containing the PORIS C++ model of ARCGenIII and its current configuration. All instrument parameters are encapsulated within that object.

This is the mockup of the event handling function, automatically generated inside the csysARC.user folder:

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

```
int ARCGenIII_cfg_init_expose(PORISNode *mynode){
  LOG(LOGINFO,'ARCGenIII_cfg_init_expose executed');
  return true;
}
```

The task of the software engineer is to convert this model into the final function, capable of initializing, configuring and commanding the detector's electronic controller, with the aim of obtaining a series of images that are saved in FITS files.

As stated before, to keep human-written code immutable against model changes, the PORIS toolkit provides a mechanism to interrogate the value of instrument parameters and modes without depending on the model architecture. This is an example of how the code written by the software engineer interrogates the model configuration to obtain one of the relevant parameters for the operation.

The example also includes how the model's OpenShutter mode is converted to a Boolean argument that is needed by the `ArcDevice_Expose()` function of the ARCGenIII driver API.

```
int ARCGenIII_cfg_init_expose(PORISNode *mynode)
{
...
  PORISSys *arcgenmodel = (PORISSys *)mynode;

  PORISSys *openshutter = arcgenmodel->getDescendantFromName('OpenShutter');
...
  PORISMode *openshuttermode = openshutter->selectedMode;
...
  if (openshuttermode->name == 'OpenShutterMode_on')
  {
    apicfg.uiOpenShutter = true;
...
    ArcDevice_Expose(apicfg.fExpTime, apicfg.binRows, apicfg.binCols, exposeCallback, readCallback,
apicfg.uiOpenShutter, &status);
  }
...
```

The `getDescendantFromName()` function allows the code to find the OpenShutter subsystem independently at its location within the model. No matter the hierarchical position of the subsystem, the code will retrieve it. The subsystem name is the unique key used to find the information. These kinds of mechanisms increase the immutability of handwritten code against model refinements.

The ability to interrogate the configuration model is not only useful for obtaining parameters to configure or control hardware. It is also useful when post-processing the results of the event. In the following example, we will see how the active mode of the Recomposition parameter is used in the post-process responsible for creating the bitmap that must be saved in the FITS image file.

```
PORISMode *mode = ObtainSelectedModeForDescendant(detector, 'Recomposition');
if (mode->name == 'RecompositionMode_None')
{
  apicfg.recomposition = RECOMPOSITION_NONE;
}
else
{
  if (mode->name == 'RecompositionMode_Serial')
    {
      apicfg.recomposition = RECOMPOSITION_SERIAL;
...
```


Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

Another example of post-processing that interrogates the model is the creation of the metadata that will accompany the images. The main purpose of this metadata is to record the conditions under which the images were obtained, such as the exposure time, the filter used, the dispersion element used, etc. The metadata can be obtained directly from the instrument configuration, available in the PORIS model of that configuration. The PORIS toolkit provides functions to dump the instrument settings directly into the image metadata, and these functions are not dependent on the instrument itself. These functions simply walk through the entire instrument setup and convert it to metadata that will be embedded in the headers of the FITS image files.

Figure 34 shows the OSIRIS detector being configured and controller by the ARCGenIII test GCS device described here.

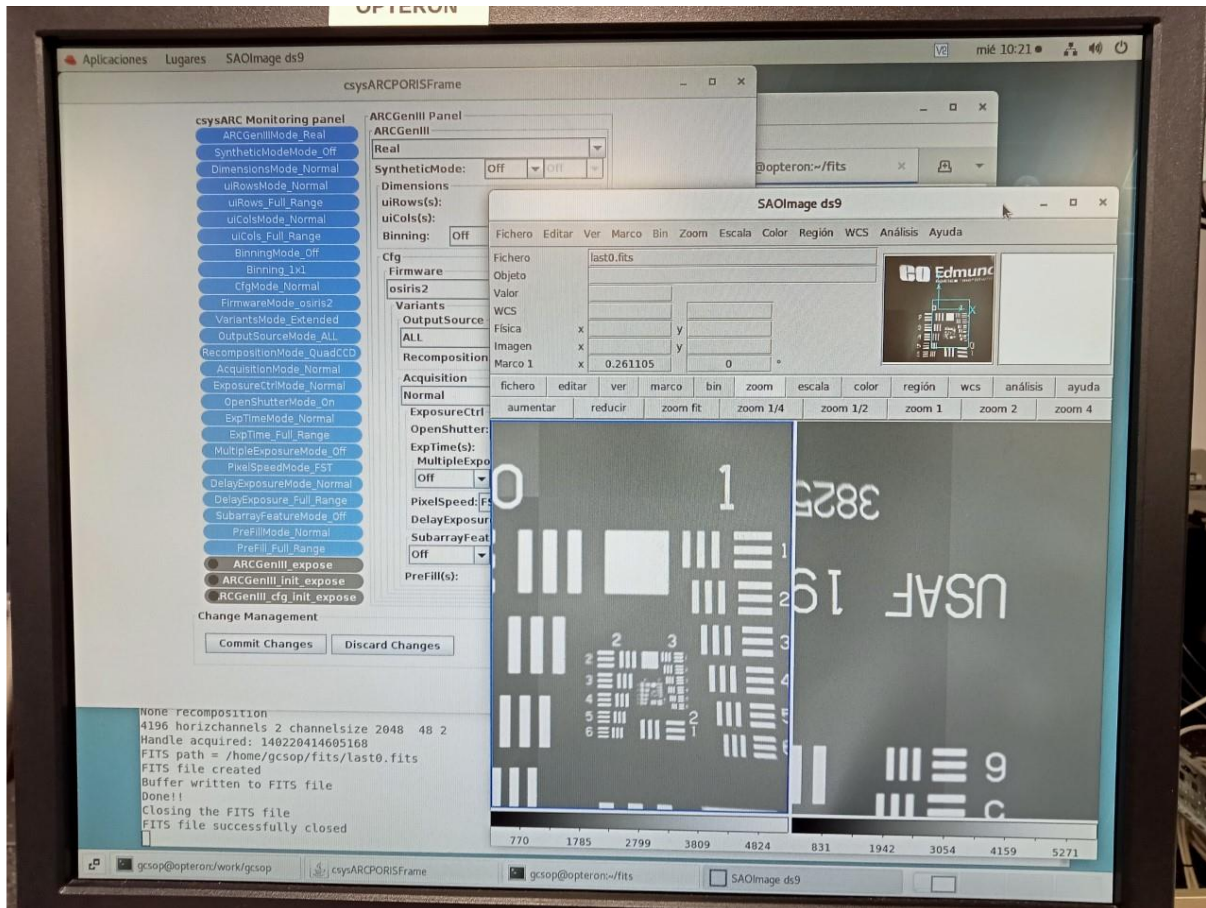


Figure 34: The new OSIRIS detector is managed by a physical GCS device generated by the PORIS toolkit, in a large percentage, automatically from the model.

To gauge the positive impact of using the PORIS toolkit, we can indicate that the number of manually written lines of code for this example (including comments in the code and using non-compact coding) is approximately 1,700.

The number of lines of code written automatically from the model, which we can consider free of typos or human errors, is greater than 20,000. The compactness of this code is comparable to that written manually.

Regarding the number of files (keeping the headers separate), the handwritten code consists of two files in a single programming language, while the automatically written code consists of about forty files that include 3 programming languages and 3 DSL.

A deeper analysis of the code in terms of complexity has not been done yet.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

4. BENEFITS OF MODELING WITH PORIS

4.1 Keeping solutions close to the problems they solve

It is relatively easy for an engineer to define a subsystem's parameters and their valid ranges and offer them to customer systems through a well-defined interface. Offering the devices in such a simple way leaves all the responsibility for checking the validity of the subsystem configuration to the systems that use it. We could say that the subsystems become simple containers of parameterized functionality at the service of whoever wants to use it, avoiding supervising that use in order to repel aberrant configurations.

In the case of subsystems that are designed to interact, the resolution of any conflict or relationship between modes and parameters of these subsystems is left to the logic of the system component that encompasses them all. In this way, each time a new dependency between subsystems is added, the complexity leaves the lower levels and rises to the higher ones.

This is causing an undesirable effect: the solution tends to move away from the problem it is trying to solve. What if the dependency is produced by some specificity of the construction of the subsystems? It is convenient that the element that solves a problem is as close as possible to the problem. If the problem must go up levels of abstraction to be solved, the knowledge related to the specificities of the subsystems must be handled at higher levels. This makes the lower-level components contain too little value, and the higher levels too smart, complex, and diverse. Beyond the conceptual problem of who must solve the problem, doing it in components far from the problem implies that the information also must be transported from the place where it is generated or consumed to the place where the solution is implemented, causing the consequent overload of communications by adding more and more information flows to the interfaces.

Let us imagine that the undesirable effects of a sensor failure in the exhaust system of a car must be solved by limiting the revolutions at which the engine must work. The vehicle is the system that includes the exhaust system and the engine and must limit the working revolutions of the latter if the aforementioned breakdown occurs. If we keep the subsystems very simple, the vehicle logic will become complicated, and sensor and throttle information must be consumed and handled at the vehicle level. The logic that commands the mitigating solution is implemented far from the subsystems involved.

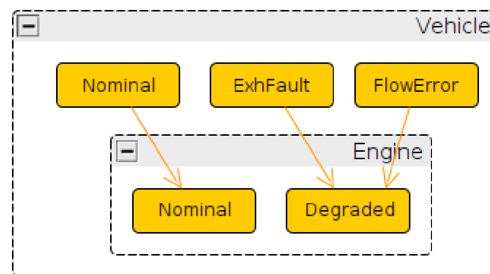


Figure 35: Example of a very simple PORIS model where modes at higher level activate modes at lower level.

One conceptual tool that engineers have at their disposal to try to contain complexity is the mode of operation. Based on this example, the motor can be asked to run in nominal or degraded mode. If the engine is in degraded mode, it will automatically apply the necessary corrections to the throttle command, adding value to the engine, and removing liability at the vehicle level. The exhaust system could be asked to emit a specific alarm to activate this degraded mode directly. The knowledge of the interrelationship between the exhaust system and the engine is contained in them. If any logic were implemented at the vehicle level, it would be very simple.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

To keep models simple and keep complexity at lower levels, PORIS forces the modeling team to think of the system in terms of modes of operation. Modes play a key role in defining a system using the PORIS language. The language forces the user to define at least one operating mode for each subsystem or parameter. While this makes modeling less intuitive, in practice it makes PORIS diagrams very easy to read. Figure 35 shows this legibility.

Speaking in terms of modes of operation, we could redefine a vehicle as a system capable of operating in various modes of operation, one of them being 'exhaust sensor failure'. In that mode, it forces your engine to operate in its 'degraded' mode of operation.

We can also perform some additional language manipulations to express the vehicle definition in terms of its configurability:

- A vehicle can be configured to work in 'nominal' mode, 'exhaust sensor failure', among others.
- The vehicle's engine can be set to run in 'nominal' or 'degraded' mode.
- 'Exhaust fault' alarms force the vehicle configuration to change from 'normal' to 'exhaust sensor fault'.

4.2 Drawing a clear border between valid and aberrant system configurations

If we understand the parameters as dimensional axes, the configurability space of a system is a multidimensional matrix whose limits are defined by the valid range of values of each parameter. The valid configurations of the system could be seen as a sparse set of regions within the matrix, surrounded by the aberrant configurations. From the point of view of the non-domain expert engineer, the location and shape of these regions appear arbitrary.

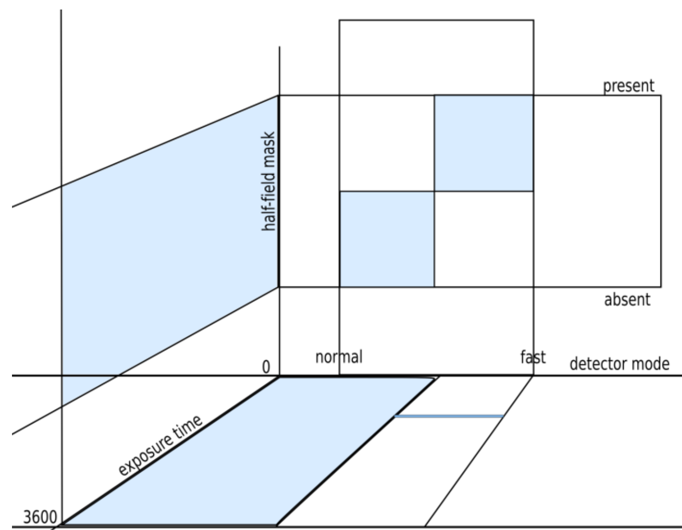


Figure 36: A representation of the relationship between three device parameters. The three-dimensional figure representing the valid configurations in this space has not been drawn because the result would not have been legible. The relationship between two of the three 2D pairs of the three dimensions has been drawn. The intersection in space of the three planes would create the 3D representation of the valid configurations.

The reason why the combination of the values of two or three parameters is considered aberrant or valid is usually found in an interdisciplinary knowledge base. In some cases, the reason for a conflict between parameters comes from the field of astronomy, in other cases from engineering disciplines: optics, electronics or mechanics, etc. It can also come from budget restrictions, or from a decision made based on the statistics of the expected use of an instrument. The reason could

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

even come from the field of what the human resources required to operate the instrument can do, or for safety and health reasons.

In summary, we can say that the configurability of an instrument must be defined by the experts of every one of the domains with which the instrument is related. That is why we consider it highly desirable that the model be readable by the full range of experts involved in the development of the instrument. From our point of view, it is much better for experts to collaborate, than to promote the skills of system engineers so that they end up being experts in all domains.

Once the system has been modeled with PORIS, the team will receive a diagram that will function as a map that clearly and unambiguously shows the border that separates the valid configurations from the aberrant ones. And furthermore, this guide is provided in a more readable format than a multidimensional array. As shown, the PORIS diagram of Figure 8 can be compared with the unfinished diagram of Figure 36 and conclusions can be drawn.

4.3 Implicit engineering modes

Another important feature of the PORIS toolkit is the automatic addition of engineering modes for all nodes of the PORIS model. This allows expert users to configure the instrument parameters in a way that the PORIS model would not have allowed, in a very controlled way, not allowing those configurations that, by definition, are aberrant, to be confused with valid ones. Without the addition of this feature, once a system is modeled and its configuration and control software are generated, there is no way to aberrantly configure it, as the model will not allow it. This strength, which protects the instrument from attempts to configure it in an aberrant way, becomes uncomfortable when, in order to investigate possible ways to improve the instrument, engineers or operators may want to explore new ways of configuring the instrument.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

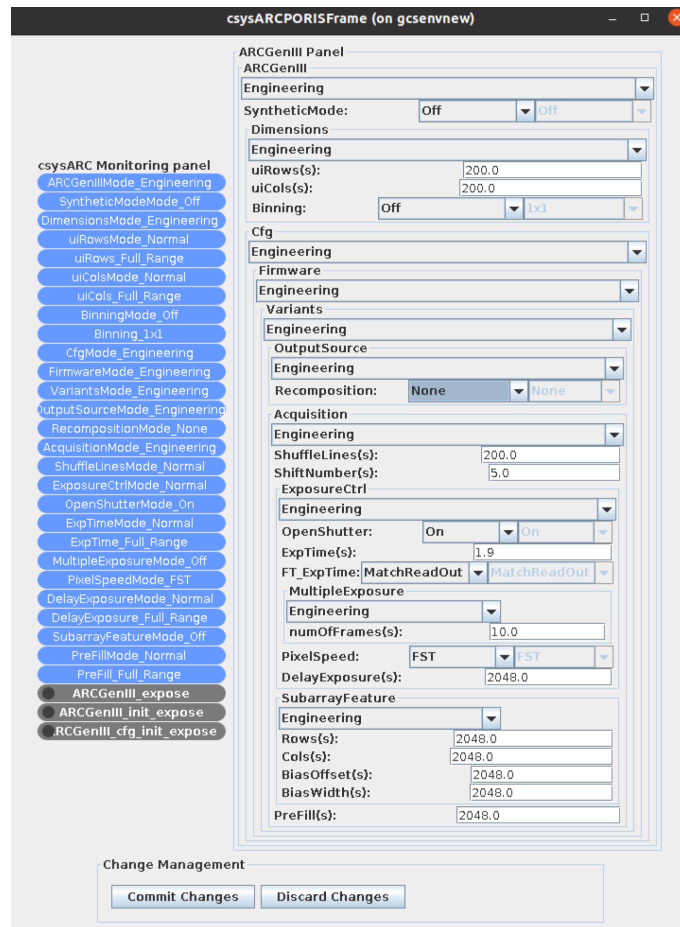


Figure 37: This screenshot shows how the ARCGenIII test GCS device can be operated in engineering mode.

Of course, the modeler could have explicitly added the engineering modes to the diagram, but this would have introduced a lot of complexity into the model, making it very difficult to read. An engineering mode in PORIS implies not only the introduction of said mode, but also the addition of a relationship with almost all the parameters or submodes within its environment. We could safely say that adding engineering modes for all subsystems and parameters of an instrument will lead to a 40% increase in the number of modes and a 200% increase in the number of diagram relationships. The reader can easily anticipate the impact of these numbers in diagrams such as Figures 17 through 22.

This feature means that every subsystem or parameter drawn in PORIS gets an additional implicit mode called 'Engineering' that nobody must add to the diagram, so nobody must maintain it in the diagram, and no documentation needs to be written for it. If a new possible value appears for a parameter, the corresponding engineering mode will automatically allow the user to select it, without drawing any new relationships.

To configure a subsystem or parameter in its engineering mode, all nodes above it must be running in engineering mode. In this way, if engineering mode is deselected at any level, all descendants will abandon their corresponding engineering modes, forcing valid configurations at their respective levels.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

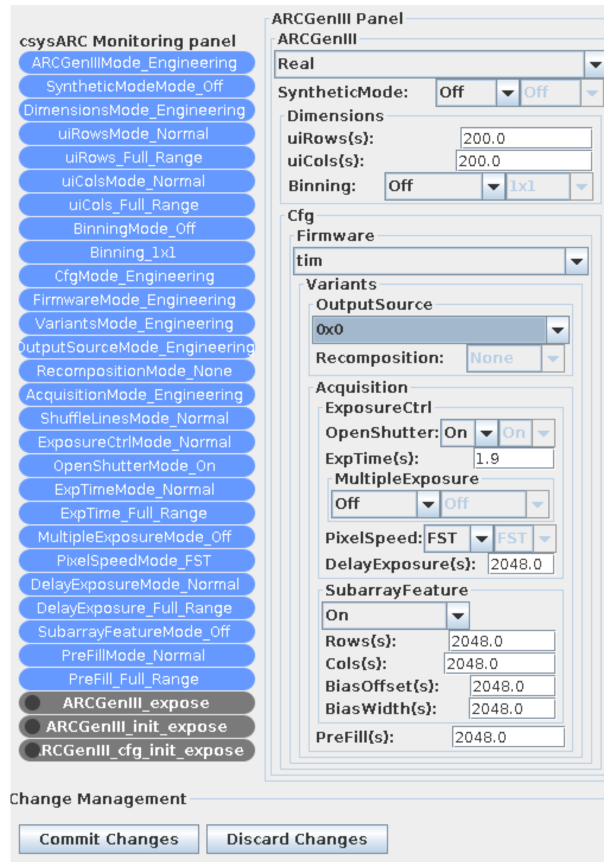


Figure 38: This screenshot shows the result of selecting and applying the 'Real' mode of the instrument while it was configured in the engineering mode shown in the previous figure. As we can see, when selecting the 'Real' mode at the instrument level, all the engineering modes of all its subsystems or parameters were deselected.

4.4 Safety supervisors

By adding the implicit engineering mode, we realized that the PORIS model had lost its function of ensuring the safety of the instrument configuration. By fundamental logic, the experts would never have modeled a valid configuration that could imply safety issues so, in practice, any dangerous configuration will necessarily belong to the space of aberrant configurations of the instrument. By protecting the instrument from aberrant configurations, PORIS is also protecting it from dangerous configurations. But the implicit engineering modes allow operators to unlock aberrant configuration space, thus making them responsible for not causing a disaster.

Let's imagine a system with two fiber optic positioning arms that can occupy the same physical position. These two arms have been designed this way to increase productivity, just like human arms: they are specialized to work on their side of the body, but both can manipulate objects that are in front of the person.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

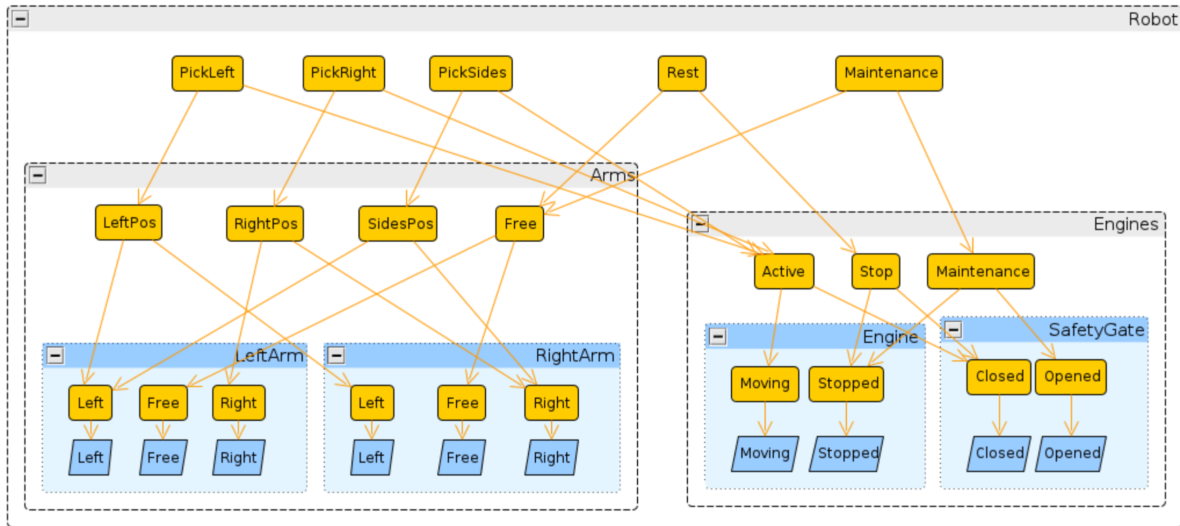


Figure 39: An example of a system where implicit engineering modes allows dangerous configurations. In this system, if the LeftArm is in the Right position and the RightArm is in the Left position, they will collide. And if the engine is running while the SafetyGuard is open, a human operator's finger could get crushed. If no engineering mode is selected, the system is safe.

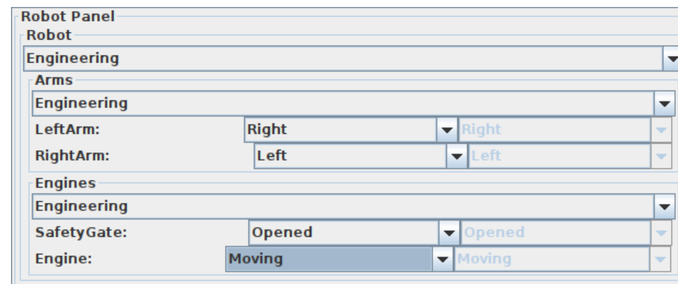


Figure 40: The configuration panel, when using engineering modes, allows the instrument to be set to a configuration that triggers both hazardous conditions at the same time.

The control software of this imaginary instrument, acting as the human brain would do, is responsible for preventing both arms from colliding. Figure 39 shows a model of such a system which, if no engineering mode is selected, guarantees that both arms will never collide.

An operator can use the Robot and Arms engineering modes to be able to set the LeftArm and RightArm to collide in the center of the work area. This shows that the safety of the instrument cannot be guaranteed from the PORIS model, due to the introduction of implicit engineering modes. Figure 40 illustrates this fact.

To correct this safety flaw, a complementary PORIS model that represents dangerous configurations may be driving the automatic implementation of a safety supervisor. As dangerous configurations are few, when compared with the total amount of possible configurations, these ‘negative models’ are very easy to model and to understand.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

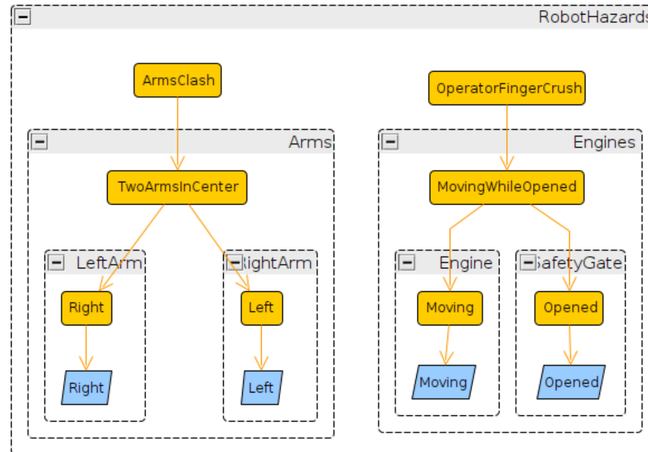


Figure 41: Negative PORIS model used to prevent Robot's engineering modes from causing safety issues.

Figure 41 shows a negative model to avoid safety problems for the system depicted in Figure 39. An interesting similarity can be seen between this diagram and the fault tree diagrams used in the safety assessment of a system. This opens a no less interesting opportunity to study the synergies between modeling configurability and designing the safety related features of a system.

4.5 Automatic documentation

The PORIS toolkit offers a tool to synchronize PORIS models in GraphML diagrams and PORIS models that live in the database of a cosmoSys instance.

The cosmoSys web application, which is a modification of the Redmine project management web application, provides a collaborative cloud environment for editing and managing PORIS models.

Thanks to this integration, the team can annotate the model enriching it with more metadata and information. Figure 42 shows a view called MainReport where the instrument's reference documentation is displayed, including diagrams illustrating the dependency relationships between its components.

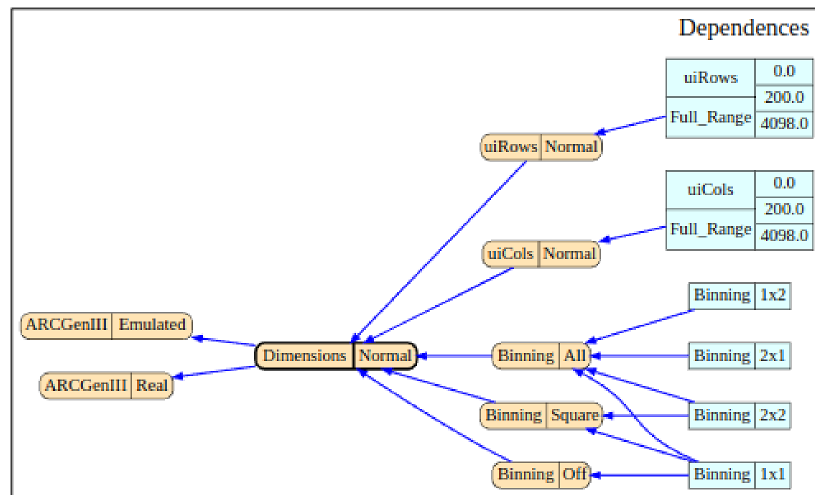
Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

1.4.: Dimensions

Componente que determina las dimensiones de la imagen a obtener, mediante los parámetros de filas (uiRows), columnas (uiColumns) y la configuración del subcomponente Binning.

1.4.1. ARC-0100: Normal

Modo normal del componente Dimensions, permite que todos sus parámetros se configuren en sus rangos completos, compatibles con el detector de Osiris.

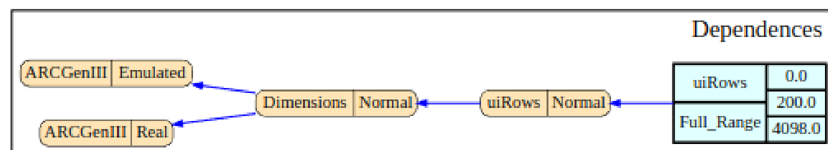


1.4.2.: uiRows

Parámetro que define el número de filas del detector que se quieren utilizar para obtener la próxima imagen. Las filas se cuentan en píxeles físicos, independientemente de la configuración de binning.

1.4.2.1. ARC-0022: Full_Range

Rango completo para el parámetro de filas uiRows. Se restringe para que no pueda ser superior al detector de OSIRIS.



1.4.2.2. ARC-0023: Normal

Modo normal de uiRows, que activa el rango normal (completo).

Figure 42: A screenshot from cosmoSys web application showing the MainReport view of the PORIS model of the ARCGenIII test GCS device. Users have annotated the model by adding textual descriptions of each model element. MainReport combines a hierarchical view of the model that mimics the style of book chapters with automatically drawn diagrams that show the dependencies of each model node.

Figure 43 shows the dependency diagram of a PORIS model, showing the propagation of configuration options based on the relationship between modes and parameters.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

Figure 43: cosmoSys dependency diagram for an instrument. The leftmost nodes represent the possible options at the instrument level, while the rightmost ones show the valid value ranges for the instrument parameters.

Figure 44 shows the subset of the dependency diagram related to a specific mode of a specific subsystem. This diagram appears when the user clicks on the node in a cosmoSys diagram.

Figure 44: The dependency diagram for the SubArrayFeature mode ‘On’.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

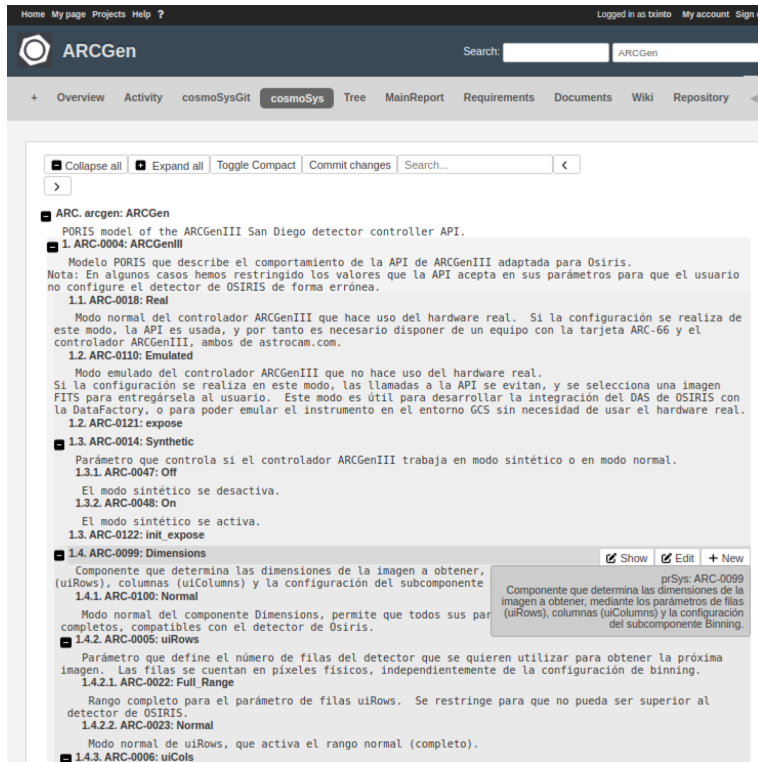


Figure 45: A tree view of the PORIS model of the ARCGenIII test GCS device. This view allows easy arrangement of the model hierarchy.

B	D	F	G	H	I	J	K	L
link	ID	subject	description	tracker	Rlv?	status	parent	blocking_items
#23	ARC-0026	1x1	Sin compresión de píxeles, ni en horizontal ni en vertical.	prValue		rqDraft	ARC-0008	
#24	ARC-0027	1x2	Compresión x2 en la coordenadas vertical, sin compresión en la horizontal.	prValue		rqDraft	ARC-0008	
#25	ARC-0028	2x1	Compresión x2 en la coordenada horizontal, sin compresión en la vertical.	prValue		rqDraft	ARC-0008	
#26	ARC-0029	2x2	Compresión x2 tanto en la coordenadas horizontal como en la vertical.	prValue		rqDraft	ARC-0008	
#27	ARC-0030	Off	Modo desactivado del binning, que fuerza el valor 1x1, sin compresión en ninguna de las dos coordenadas.	prMode		rqDraft	ARC-0008	ARC-0026
#28	ARC-0031	All	Modo completo del binning, que permite al usuario escoger cualquiera de los cuatro valores de compresión.	prMode		rqDraft	ARC-0008	ARC-0026,ARC-0027,ARC-0028,ARC-0029
#29	ARC-0032	Square	Modo cuadrado del binning, sólo permite valores de binning que sean iguales en la vertical y la horizontal. * 1x1: ninguna compresión. * 2x2: compresión en ambas coordenadas.	prMode		rqDraft	ARC-0008	ARC-0026,ARC-0029
#101	ARC-0101	Cfg	Componente principal de configuración de ARCGenIII. Permite escoger variantes (versiones de firmware) y maneja los parámetros que dependen de esa elección.	prSys		rqDraft	ARC-0004	
#102	ARC-0102	Normal	Modo normal del componente de configuración, que permite escoger todas las versiones de firmware.	prMode		rqDraft	ARC-0101	ARC-0021,ARC-0062,ARC-0073,ARC-0074,ARC-0075
#4	ARC-0007	Firmware	Componente de firmware, que permite escoger la versión de firmware que se cargará en el controlador.	prSys		rqDraft	ARC-0101	
#18	ARC-0021	tim	Versión oficial de firmware de ARCGenIII, desarrollada por astro-cam. No incorpora los modos especiales desarrollados en LISA para Osiris.	prMode		rqDraft	ARC-0007	ARC-0098
#62	ARC-0062	osiris2	Primera versión del firmware de ARCGenIII que da soporte a los modos especiales desarrollados en LISA para Osiris.	prMode		rqDraft	ARC-0007	ARC-0105
#73	ARC-0073	osiris3	Segunda versión del firmware de ARCGenIII que da soporte a los modos especiales desarrollados en LISA para Osiris. Nota: en la actualidad usa el mismo fichero de firmware que la opción osiris2.	prMode		rqDraft	ARC-0007	ARC-0105
#74	ARC-0074	osiris4	Tercera versión del firmware de ARCGenIII que da soporte a los modos especiales desarrollados en LISA para Osiris. Nota: en la actualidad usa el mismo fichero de firmware que la opción osiris3.	prMode		rqDraft	ARC-0007	ARC-0105
#75	ARC-0075	osiris5	Cuarta versión del firmware de ARCGenIII que da soporte a los modos especiales desarrollados en LISA para Osiris. Nota: en la actualidad usa el mismo fichero de firmware que la opción osiris4.	prMode		rqDraft	ARC-0007	ARC-0105
#97	ARC-0097	Variants	Componente que engloba las variantes de parámetros que se incluyen en las diferentes versiones de firmware.	prSys		rqDraft	ARC-0007	
#98	ARC-0098	Normal	Modo que selecciona la variante Normal de parámetros, que da servicio al firmware tim.	prMode		rqDraft	ARC-0097	ARC-0077,ARC-0087,ARC-0088,ARC-0089,ARC-0090,ARC-0091
#340	ARC-0105	Extended	Modo que selecciona la variante Extendida de parámetros, que da servicio a los diferentes firmware osirisX.	prMode		rqDraft	ARC-0097	ARC-0077,ARC-0078,ARC-0079,ARC-0087,ARC-0088,ARC-0089,ARC-0090,ARC-0091,ARC-0092

Figure 46: The cosmoSys export file format allows the team to read or modify the model in offline mode. This is especially convenient to support revision processes, to store model baselines, or to support bulk edits of textual model attributes.

Figures 45, 46, 47 and 48 show other examples of documentation generated automatically from cosmoSys. Many of these cosmoSys features are described in the article at Ref. 4.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

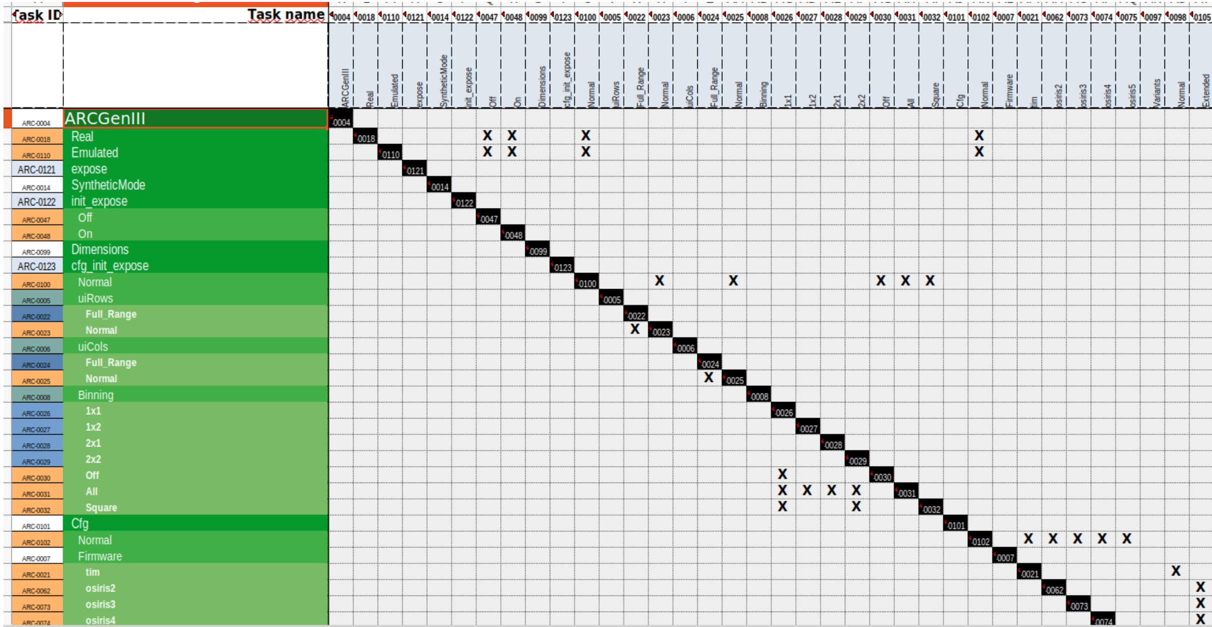


Figure 47: The cosmoSys export file format also includes other useful views. In this case, a DSM (Design Structure Matrix) view shows the dependencies between the model nodes.

COSMO BOTS	ARCGen **Code:** REQ_ARC { }	**Page:** 5 of 53 **Date:** 01/01/00 **File:** REQ_ARC

TABLE OF CONTENTS

Table of Contents

1. ARC-0004:ARCGenIII....10
1.1 ARC-0018: Real....10
1.2 ARC-0110: Emulated....11
1.3 ARC-0121: expose....11
1.4 ARC-0014: Synthetic....12
1.4.1 ARC-0047: Off....13
1.4.2 ARC-0048: On....13
1.5 ARC-0122: init_expose....14
1.6 ARC-0099: Dimensions....14
1.6.1 ARC-0100: Normal....15
1.6.2 ARC-0005: uiRows....16
1.6.2.1 ARC-0022: Full_Range....16
1.6.2.2 ARC-0023: Normal....16
1.6.3 ARC-0006: uiCols....16
1.6.3.1 ARC-0024: Full_Range....17
1.6.3.2 ARC-0025: Normal....17
1.6.4 ARC-0008: Binning....17
1.6.4.1 ARC-0026: 1x1....18
1.6.4.2 ARC-0027: 1x2....18
1.6.4.3 ARC-0028: 2x1....18
1.6.4.4 ARC-0029: 2x2....18
1.6.4.5 ARC-0030: Off....18

Figure 48: An example of the table of contents of a report written from the PORIS model of the ARCGenIII test GCS device.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

4.6 Partitioning and absorbing complexity

The dynamic configuration of a system changes the expected behavior of the system when processing its inputs and its internal state to calculate its outputs.

Thinking of the system and all its component subsystems as operating in modes, and letting the model take on the job of propagating mode changes through the model hierarchy, from higher to lower levels, will have the effect of eliminate many of the decisions that engineers must implement in their algorithms.

This will allow engineers to go from having to develop large systems capable of considering many inputs to analyze the execution context and make decisions about how the system should act at each moment, to simply having to develop small functional cells that will be activated, or they will be deactivated depending on the modes and values of the parameters that make up the active instrument configuration at a given moment.

This is a good strategy because we must remember that any decision modeled in a PORIS diagram has a direct support link with the multiple experts that support that decision. And that decision will enjoy the benefits of automatic documentation and easy refinements that decisions that have been confined within traditionally developed functions lack.

We will use the example of a software module that performs closed-loop control tasks, calculating the action an actuator should take based on its own internal state, some set point, and the information it collects from various sensors.

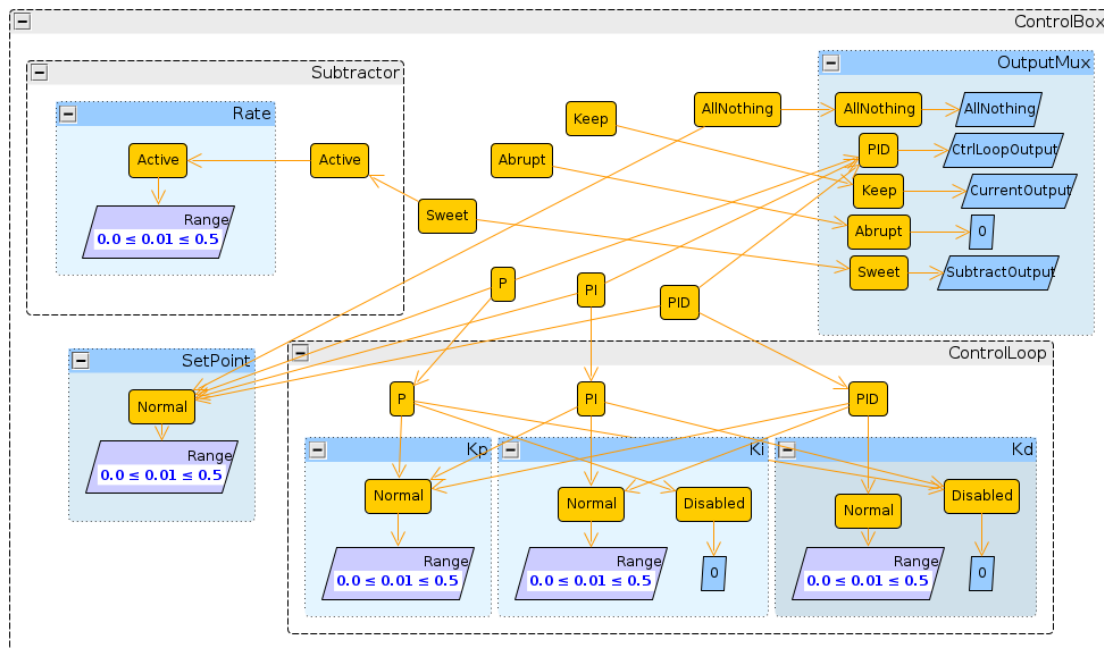


Figure 49: ControlBox model. It was drawn, and its control panel generated in just 17 minutes

That module must be able to execute several different control algorithms: ‘AllNothing’, ‘P’, ‘PI’ and ‘PID’, and it must be able to stop in three ways called 'Keep', 'Abrupt' and 'Sweet'. The 'Keep' stop freezes the current output, stopping the control

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

loop but not changing the action sent to the actuators. The 'Abrupt' stop immediately sets the exit action to zero and keeps it there. The 'Sweet' stop gradually reduces the output action, according to a preset rate, until it reaches zero.

The model shown in Figure 49 has been drawn in a few minutes. The source code created automatically from this model includes a function called `setControlBoxMode()`, whose values will be `AllNothing`, `P`, `PI`, `PID`, `Keep`, `Abrupt`, and `Sweet`. It will also offer commands to set the control box parameters: `setRate()`, `setSetPoint()`, `setKp()`, `setKi()`, `setKd()`, `setOutputMax()` All of these calls will be available through the instrument interface.

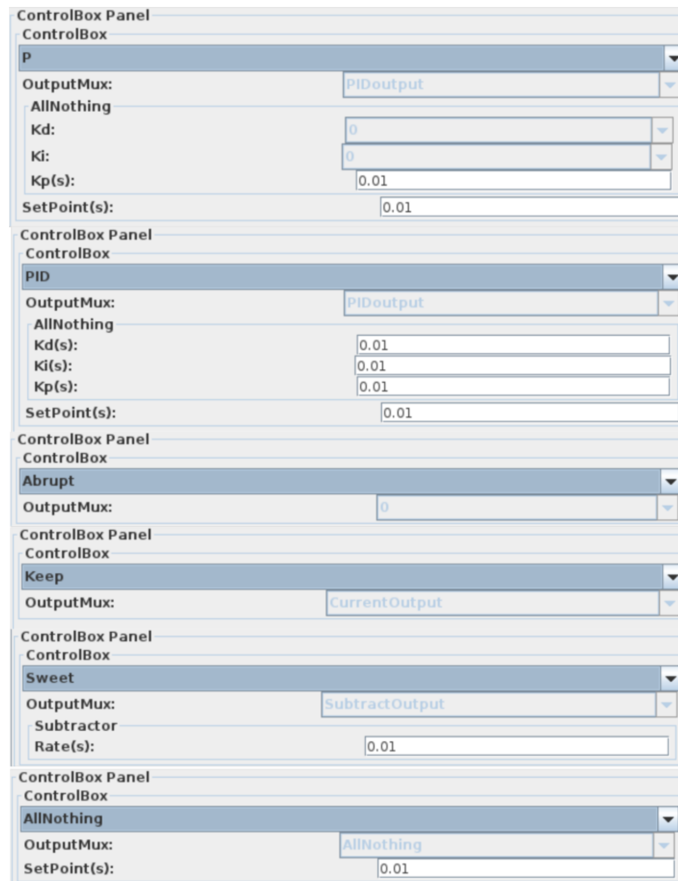


Figure 50: Screenshots of the ControlBox configuration panel showing all possible modes of operation.

Figure 50 shows the GUI to configure each of the modes. To develop the algorithms of the control box, since its complexity has been absorbed by the model diagram thanks to the modes of operation, the engineer only must implement simple algorithms, and an equally simple dispatch algorithm, which selects the algorithm to run from the values of some configuration parameters.

At diagnostic or debugging time, engineers don't have to figure out what the variables relevant to their analysis are called, because the names have already been established since the PORIS model was created.

At this point, we want to draw the reader's attention to a few facts:

- The software engineer is no longer responsible for most of the code that collects or transports or receives the parameters, checks their consistency, and verifies their compatibility with the active instrument configuration.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

This means that your task has gone from developing a complex software module that must take into account a large number of constraints, to developing only four function cells, which we define as: (a) the ControlLoop algorithm, (b) the Subtractor algorithm and the (c) OutputMux algorithm, and (d) the algorithm dispatcher, which we can categorize as (a) a library math formula, (b) a simple decrement instruction, (c) a simple multiplexer that is implemented with the typical 'switch-case' construction, and (d) a fairly simple dispatcher which, again, can be implemented with a simple 'switch-case'. There is no need to code any type of interrelation or communication between the algorithms. The simpler the code, the easier it is for it to be developed within quality standards.

- The developer is provided with a set of tools to verify and validate not only the module, but the integration with the system. The automatically generated code of the model acts as a software mockup and can be used to run your first tests. As we can see in Figure 50, the final output action is not calculated by the PORIS model, but we can use the OutputMux parameter labels as the expected result, which is enough to validate the integration with systems using ControlBox.
- A consistent, domain-centric, semantic nomenclature of variables has been defined to software developers, which is the same one used ubiquitously in PORIS diagrams and configuration panels, thereby keeping communication open and consistent. fluid among software engineers, system engineers, and domain experts.
- And finally: we have protected the handwritten code from most future changes expected from system specification improvements. A PID algorithm, a decrement, an all-nothing algorithm will hardly change, because all three are not custom software developed for the system. Its parameters (Kp, Ki, Kd, SetPoint, Rate…) are also expected to be stable, since they are part of the algorithms.

We can conclude that the use of a PORIS model has split the complexity of the system, and that the automatic PORIS products are absorbing the largest and most repetitive part of the complexity, by resolving parameter dependencies and mode propagation. In this way, it also closes the door to a whole series of development errors that can come from typographical errors as well as from errors in the copy and paste operations of the designs or codes. Also, no less important, it reduces the number of communicative errors derived from the lack of following the semantic conventions established in the project.

4.7 Jumping from configurability to control of an automatic system

In this section we present an example to understand how a PORIS model can be used as a starting point to develop the control software of a device that must govern itself, making decisions autonomously. This experience should be taken as a proof of concept of how we can extend the benefits of developing the configurability of a system modeled with PORIS towards the development of a control system for the same system.

The concept that allows transforming a control problem into a configurability problem is shown in Figure 51. Once transformed into a configurability problem, the solution can be automatically implemented starting from a PORIS model.

As an illustrative example, let's consider an electric car that must deal with 'idling', 'charging' and 'driving' situations. These situations can be categorized as states, operational modes, or missions, it just depends on a terminology decision. To avoid confusion, we have called them 'missions'.

An autonomous system acts in several ways depending on the 'mission' that it must solve at each moment. Those missions will be modeled in PORIS language as modes of operation that will be automatically selected by an algorithm executed by the system itself.

In our example, the car is not waiting for the user to arbitrarily select the mission, the choice is made automatically by the car itself, following a predefined logic, probably defined by a finite state machine (FSM).

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

Our autonomous system will have to be continuously executing the algorithm that, depending on the state of the system and the environment, chooses the mission to follow. Missions will be modeled as higher-level modes of the PORIS model.

The configuration of the system is subordinate to the mission that is being executed. The mission is built as a mode that should only allow one possible option for each of its submodes. All other modes in the system should be built, whenever possible, using the same rule. If this were possible, simply selecting the mission would be very easy to obtain the actions that the system performs. But normally this is not possible, so the system must be able to make secondary decisions within the same mission, usually based on how it perceives its state and its environment through sensors.

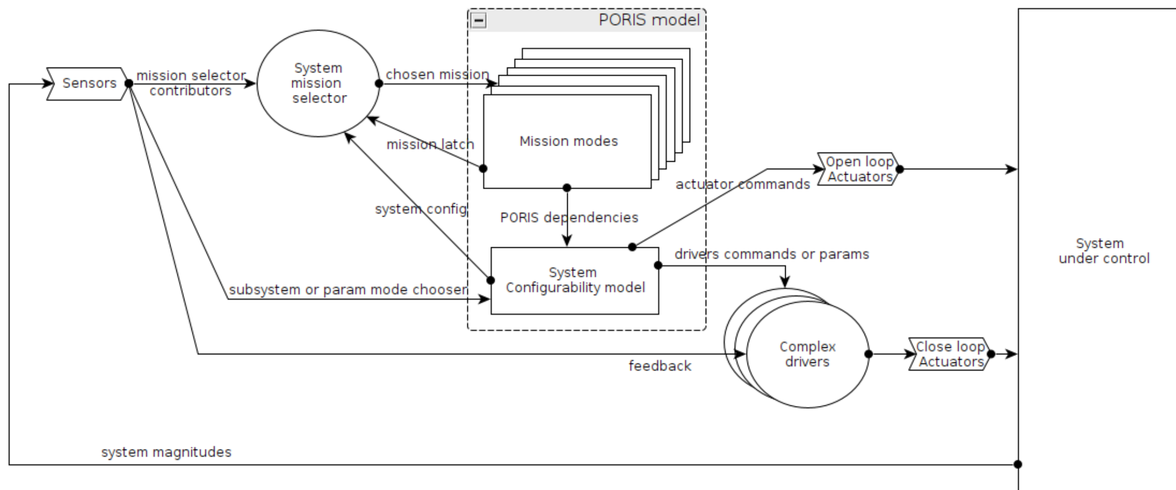


Figure 51: Converting a PORIS model into a control system

In some cases, following the rule is not possible or convenient. Some missions can need additional decisions to be taken, to avoid creating as many mission replications as possible choices exist in the secondary decision. In those cases, in which some subsystems or parameters of the system configurability model can allow several selectable modes for the same mission, the decision must be taken automatically from the system environment status. No human intervention is allowed because a control system must be able to work autonomously. Normally the value of a sensor will work as the system or parameter mode chooser.

The actuators can be managed directly from the decision made by the PORIS model, or they can be complex to drive, requiring an additional functional cell that complements that decision. A decision to 'lock' or 'release' a door, modeled as possible values a parameter of the PORIS model, can easily be translated into a command from the door lock actuator, but a decision like 'keep the detector at 100ºK' will require an additional complex controller.

The system status, made up of the system magnitudes read from the sensors, the current system configuration, and the current mission, is the input to the system's mission selector, which will choose the next mission to perform. It is quite normal for the PORIS model to implement part of the system's mission selector, using a virtual subsystem to select the next mission. In figures 54, 55 and 56 of the MissionRobot example we will see how a virtual subsystem called NextStep is the one that decides that the time has come to change the mission.

The example consists of a line-guided toy robot that must execute a path, following a black line, deciding which branch of the path to follow based on a predefined list of options. Figure 52 illustrates the path.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

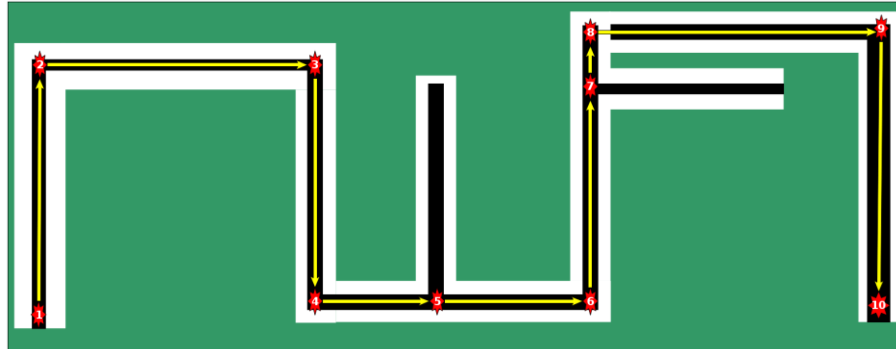


Figure 52: The path that the MissionRobot must follow, including decision points. These decision points break the path into fragments. The fragments will be used to define the robot's missions.

In this example, each numbered fragment of the path will be an entry in the mission list. Since the fragments are sequential, the mission list is best understood as a mission queue.

The choice of the next mission is binary: should the system continue with the next mission, or should it continue executing the current mission? The response will be a mode of the NextMission subsystem, which can be 'Yes' or 'No'.

We will see how the actuators (robot wheels) will be managed in a hybrid way: depending on the configuration in which the system is found, they will be commanded directly from the PORIS model, or their control will be delegated to a more complex secondary controller: a line follower.

Figure 53 shows a model for the robot's Mobility subsystem, which processes four commands: Stop, Forward, TurnLeft, TurnRight, SoftLeft, and SoftRight. SoftX moves differ from TurnX ones in that neither wheel moves backwards, resulting in a less abrupt change of direction.

Unfortunately, once the mission (fragment of the route) has been chosen, the robot cannot autonomously choose the 'Mobility' mode to use. The robot must observe how it is placed on the line to decide if it is well centered on it, if it should move forward, turn, or if the fragment that follows has reached its end. Once the robot builds a state word from its environment, knowing the mission it is executing, it can make a mobility decision.

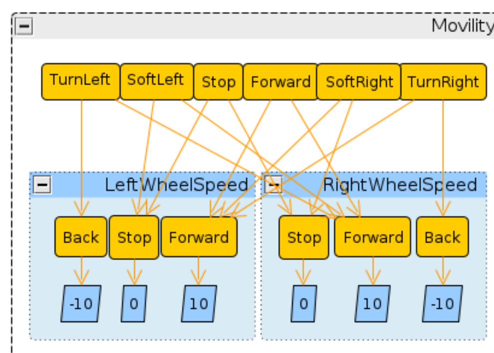


Figure 53: Model of the MissionRobot 'Mobility' subsystem

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

The robot has three photoreceptors that look at the ground, to measure its position relative to the black line it is following. A function will take all three values and calculate a word that will be used to make the decision what to command the wheels, but this decision is limited to the context of the current fragment (mission) that the robot is executing.

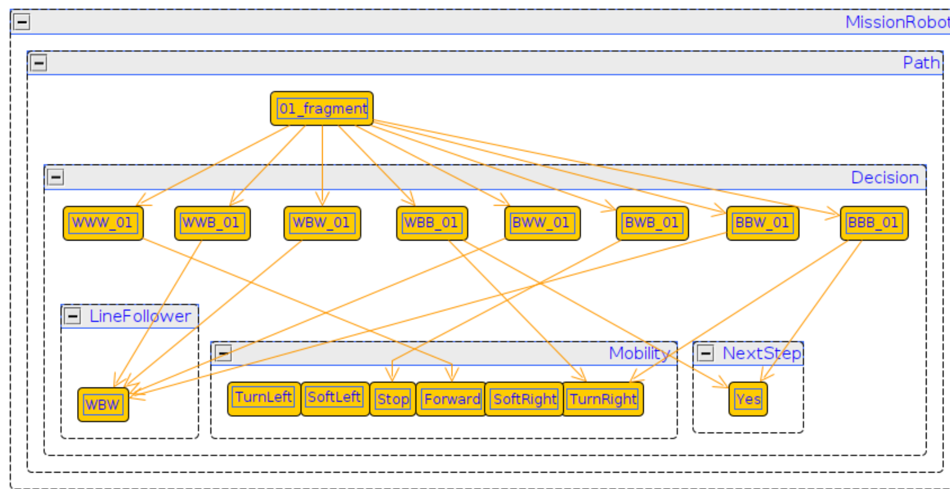


Figure 54: MissionRobot Decision model for fragment #1

Figure 54 shows how these secondary decisions can be modeled for the first fragment using the PORIS language. The decision tree can be read as:

- If the sensor word (a concatenation of the color seen by each of the three photoreceptors) is WWW (white-white-white, the robot sees no line), the robot will move forward.
- If the sensors word is WBW (white-black-white, the robot is centered on the line), then the line follower algorithm will command the wheels, with WBW being its set point.
- If the sensor word is WWB, BWW or BBW (the robot is almost centered on the line), the line following algorithm must also be activated with the WBW setpoint.
- If the sensor word is BWB (the robot is seeing two separate lines), then perhaps the robot has reached the end of the fragment and has a spurious sensor reading before BBB or WBB, or there may have been a sensor error, or maybe the robot got out of the way. Being at the beginning of the development, the experts recommend stopping the robot, since it will allow us to detect when these situations occur when debugging the guidance logic.
- If the sensor word is WBB, the end of the fragment has been detected. The robot shall turn right and move on to its next mission: fragment #2.
- The sensors word BBB will be processed as WBB, because the robot could have found the end of the fragment while slightly to the right of the line. The robot will turn right and move on to its next mission: Fragment #2.

As we can see, NextStep is set to Yes when the robot perceives its environment as consistent with the curve at the end of the fragment.

Looking at Figure 52 we can see that fragment #2 also ends when the path turns 90º to the right, so its decision tree is the same as fragment #1. As we'll see later, we'll simply add twice the fragment#1 to the mission queue. Fragment #2 will not be modeled. Fragment #3 ends with a curve to the left, so we build the model in Figure 55.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

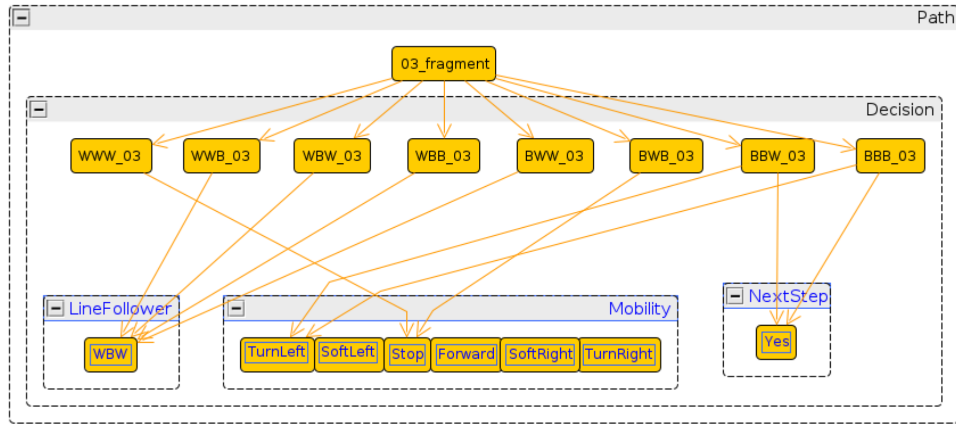


Figure 55: MissionRobot Decision model for fragment #3

The models of the rest of the path fragments will not be shown in this article, but the reader can easily infer them.

To stop the robot after executing the last fragment, a special fragment numbered #0 is modeled, and is shown in Figure 56. This fragment also allows to set the robot in a disabled mode.

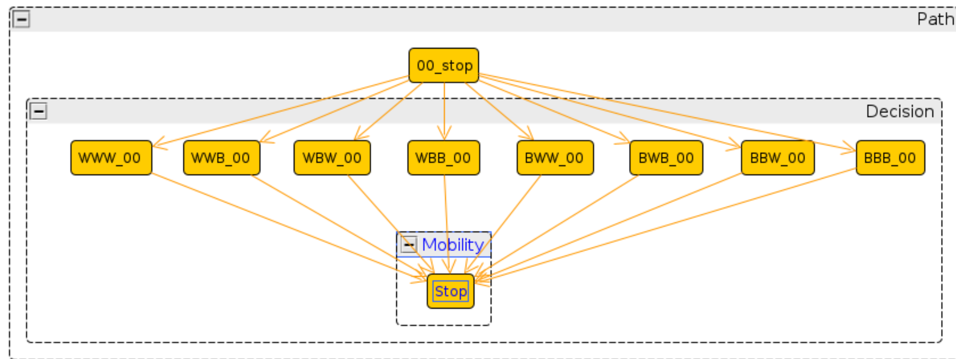


Figure 56: MissionRobot Decision model for fragment #0

We can see how the complexity can be divided into several diagrams, one per mission. The way the logic solves the problem is supported by experts, who continue to agree with the diagrams, and benefits from the entire documentation environment and automatic code generation.

To complete the robot example, this is all the Python code that required human coding, totaling less than 100 lines. The line follower algorithm is not included here:

```
from csysMissionRobotPORIS import *
...

# Motor and sensor initialization
leftMotor = Motor('A')
rightMotor = Motor('B')
```


Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

```
leftSensor = ColorSensor('C')
centSensor = ColorSensor('E')
rightSensor = ColorSensor('D')

# Obtain a letter for a given color
def colorletter(cstr):
    if cstr == 'black':
        return 'B'
    if cstr == 'white':
        return 'W'
    else:
        return '-'

# Creating a sensor word, we need it to select the decision mode acting at each moment
def sensorword():
    return
colorletter(leftSensor.get_color())+colorletter(centSensor.get_color())+colorletter(rightSensor.get_color())

# This executes the line follower algorithm based on the current model configuration
def executeLineFollower(selectedMode):
    # We skip showing this function body in the article
    ...
    # Returning the calculated speeds for the wheels
    return leftWS,rightWS


# This instantiates the Python class generated by the model
model = csysMissionRobotPORIS()

# Setting the fragments order using the handler the Python class offers
fragmentspath = [
model.md01_fragment,
model.md01_fragment,
model.md03_fragment,
model.md04_fragment,
model.md03_fragment,
model.md06_fragment,
model.md01_fragment,
model.md01_fragment
]

# Some initialization
currentFragment = 0

# To launch the robot mission, we select the normal mode for the mission
model.root.setMode(model.mdMisionNormal)

# fragments iteration
for tr in fragmentspath:
    newFragment = False
    model.sysPath.setMode(tr)
    while not newFragment:
        # Building the next decision mode to execute by joining the
        # sensor word with the fragment number (f.i. 'WWW' + 3 => 'WWW_03')
        d = sensorword() + '_' + str(currentFragment).zfill(2)
        if d in tr.submodes.keys():
            # The mode chosen by the sensor word exists
            # We select it just using its name as a key in the dictionary
            # of the submodes of the current fragment mode
            dr = tr.submodes[d]
            # Applying the new mode
            model.sysDecision.setMode(dr)
```


Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

```
            # In the case a linefollower is active, use its output
lfmode =  model.sysLineFollower.selectedMode
            if lfmode != model.mdLineFollower_UNKNOWN:
              leftWS,rightWS = executeLineFollower(lfmode)
              leftMotor.start(leftWS)
              rightMotor.start(leftWS)

            else:
                # Otherwise the wheels must follow what the PORIS configuration has chosen
                leftMotor.start(model.LeftWheelSpeed.selectedValue.data)
                rightMotor.start(model.RightWheelSpeed.selectedValue.data)

            np = model.sysNextStep.selectedMode
            if (np == model.mdNextStepYes):
                newFragment = True

    currentFragment += 1
...
```

5. ADDING LANGUAGES TO THE MODELING ECOSYSTEM

Figure 57 shows a proposal of how to include additional languages for the definition of the requirements in a project. We consider the requirements management process as a good simplification of the problem of integrating several modeling languages in the same project.

History has shown that text-based requirements can represent almost every aspect of a system. Text-based requirements can drive any development. The problem with these types of requirements is that they need to be interpreted by a human, and therefore are not useful for generating automatic products from them.

It is quite easy to automatically generate text requirements from requirements written in a formal language. This allows a language such as SysML to play a central role in a model-based engineering project: by simply converting the model to text, requirements expressed in SysML will be able to trigger the same processes as text requirements, and furthermore, it will be able to drive automatic processes.

The effort to implement synchronization mechanisms between SysML and text-based requirements can be minimized if the modeling team follows a few conventional rules. For example, if a requirement is stated in the SysML model, the associated text-based requirement should be considered a 'read-only' courtesy view for humans. Following this simple convention, the modifications of those requirements will always be made on the SysML model and will be automatically propagated to text without running the risk of losing information. We can define this synchronization as acting in one direction only.

It is in the project's interest to keep the text-based requirements for an additional reason: the easiest way to sign a contract between two institutions is to sign a document understandable to all parties. Executive staff and experts at both institutions must understand what they are signing, and SysML is difficult for them to read. The SysML model can be easily translated into a text document, to which requirements that have been directly defined as text-based would be added.

SysML is less fuzzy than natural language, it is a more specific language than the latter. Domain specific languages (DSLs) are much more specific to the aspect of the system they are representing.

The effort to build a synchronization mechanism between a DSL and SysML, if wise conventions are followed, should be very small compared to the benefits to be gained. Since both languages are formal, two-way synchronization can be achieved. Automated products should be generated from the SysML model to maximize the effort spent on developing automated encoders.

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

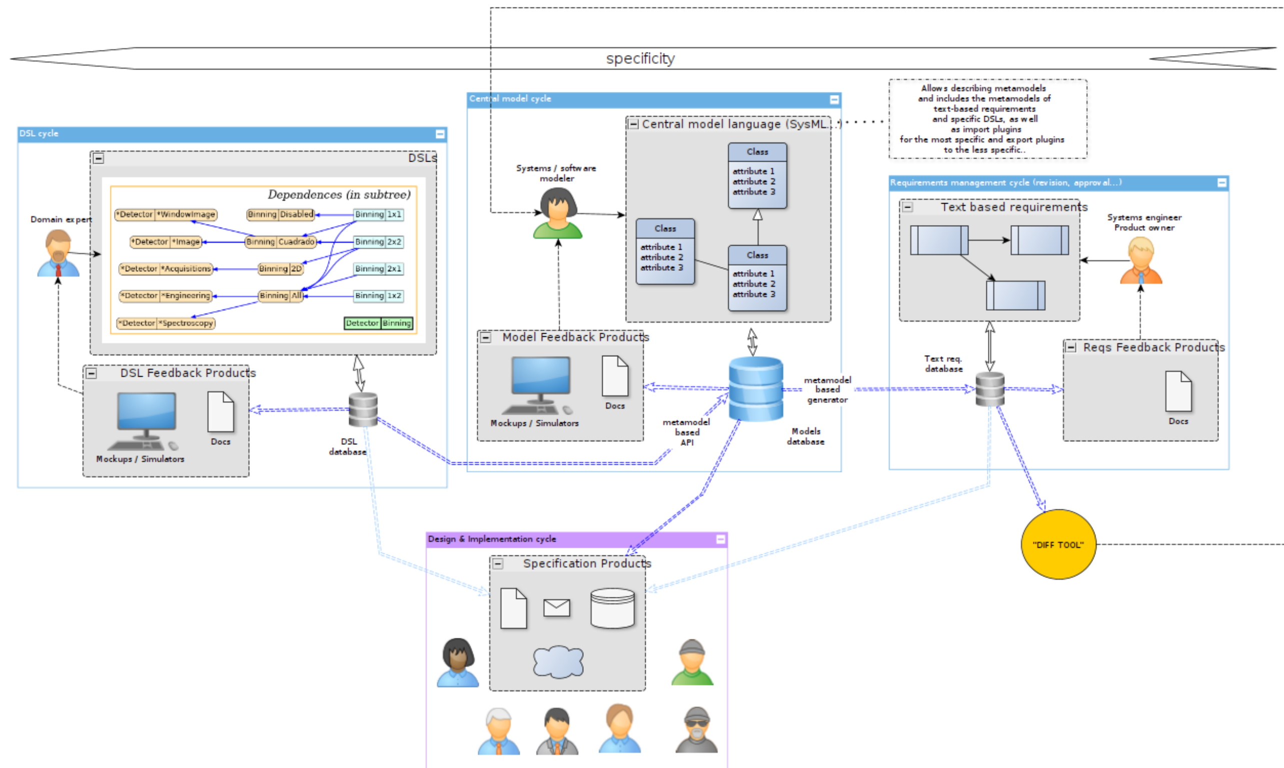


Figure 57: Collaboration between DSL modeling languages, central modeling languages, and text-based requirements. The PORIS language is included in the DSL category.

In some cases, it may be convenient to generate code directly from the DSL model, without having to go through SysML. This article has shown how experts can immediately refine PORIS diagrams by translating them directly into configuration panels, for example. In the case of the automatic GCS device encoder, it would have been convenient if the devices were generated from the SysML models. Calendar issues were the only reason we are generating the code directly from the PORIS models.

6. CONCLUSIONS

In this article, we have used our experiences with the PORIS toolkit as a proof of concept to demonstrate how cost-effective it can be to add simple modeling languages to a project.

PORIS was designed to model the configurability of a system, which we considered to be a key aspect during development, as it allows the team to define a clear border between valid and aberrant configurations of a system.

The ability to represent the configurability of a complete instrument in a few, simple, and understandable diagrams allows the team to better understand the function that the system to be developed must encompass. The PORIS toolkit, capable of automatically generating products from the model, supports the specification process by providing instant feedback on the definition of that product. Being easy to understand, the PORIS model of an instrument keeps the team of experts connected to the model.

We added selected experiences to show the additional benefits of using a simple language like PORIS in a project. The toolkit allows user to automatically implement a large percentage of the configuration and control of an instrument, directly

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use

from the model. We have shown how from the PORIS model we can automatically create a virtual device that runs within the GCS network of a telescope, and how easy it is to convert it into a physical device capable of handling hardware.

Aside from the productive benefits, the challenges that humans must face while developing a product have diminished: a large percentage of repetitive, quantitative, boring implementation effort has been stolen from people's to-do list. The code generated from the model is responsible for managing the operating modes of a system, its parameters and the dependencies between them. Much of the decision dispatching and much of the logistics happens automatically, leaving it up to people to implement inherently custom code that includes technical value, may have life beyond the current project, and is attractive to develop. The use of models also helps the human developer to keep their manually written code immutable during project development.

After showing the experiences, the article shows a proposal of how to add these simpler languages in the ecosystem of an engineering project.

ACKNOWLEDGMENTS

The PORIS language and its toolkit have been developed thanks to the support of the members of CosmoBots, with no additional funding. The PORIS GCS code generators have been developed thanks to the financial support of GRANTECAN, and the synergies with LISA detectors laboratory. The authors wish to thank the GTC and OSIRIS teams, the IAC instrumentation area staff because all they trusted in the novel engineering techniques described here.

REFERENCES

[1] Vaz-Cedillo, J.J., "PORÍS: practical-oriented representation for instrument systems," Proc. SPIE 7738, Modeling, Systems Engineering, and Project Management for Astronomy IV, 77381W (5 August 2010); https://doi.org/10.1117/12.856262

[2] Vaz-Cedillo, J.J., Bongiovanni, A.M., Ederoclite, A., et al., "The multi-object spectroscopy (MOS) observations automatized production line," Proc. SPIE 10705, Modeling, Systems Engineering, and Project Management for Astronomy VIII, 107050X (10 July 2018); https://doi.org/10.1117/12.2312730

[3] Cepa, J., Aguiar, M., Escalera, V. G., et al. 2000, "OSIRIS tunable imager and spectrograph", in Proc. SPIE, Vol. 4008, Optical and IR Telescope Instrumentation and Detectors, ed. M. Iye & A. F. Moorwood, 623–631

[4] Vaz-Cedillo, J.J., González-Rocha, R., et al. "cosmoSys-Req: A free open-source requirements management tool" Paper 12187-56 (07 2022)

[5] Booth, J.A., Crause L.A., "How to talk so your engineer will listen, how to listen so your scientist will talk: the human side of astronomical instrument development," Proc. SPIE 10705, Modeling, Systems Engineering, and Project Management for Astronomy VIII, 1070509 (10 July 2018); https://doi.org/10.1117/12.2314087

[6] López-Ruiz, J.C., Vaz-Cedillo, J.J., et al., "A mask quality control tool for the OSIRIS multi-object spectrograph," Proc. SPIE 8451, Software and Cyberinfrastructure for Astronomy II, 84511Q (24 September 2012); https://doi.org/10.1117/12.925682

Downloaded From: https://www.spiedigitallibrary.org/conference-proceedings-of-spie on 12 Sep 2022
Terms of Use: https://www.spiedigitallibrary.org/terms-of-use