SIMPLIFYING FLIGHT TEST CONFIGURATION WITH CONSTRAINTS

Patrick J. Noonan Jr., Hakima Ibaroudene,
Austin J. Whittington, Myron L. Moodie
Southwest Research Institute®
San Antonio, Texas

patrick.noonan@swri.org, hakima.ibaroudene@swri.org,
austin.whittington@swri.org, myron.moodie@swri.org

ABSTRACT

Configuring flight test systems can be a complex process due to the large number of choices that must be made. Making these choices requires system knowledge to build a working configuration in an efficient and timely manner. Historically, flight test systems have embedded this system knowledge in code. The limitation with these approaches is that any change or addition to the system knowledge is costly due to the significant work required to update and maintain the software. We see the philosophy of constraints as a promising path toward addressing these issues. In the context of flight test configuration, a set of constraints defines the limits of how a system may be configured to perform specific tasks. This paper describes an approach for simplifying configuration by moving the system knowledge out of hardcoded business rules and into a flexible architecture that leverages constraints for validation of system configurations.

KEYWORDS

Flight Test Instrumentation, Constraints, MDL, XML

INTRODUCTION

The current telemetry component provider landscape is diverse in approach and capability, and this landscape continues to evolve as new test articles require new components, which leads to new system management requirements and complexities, especially in the area of configuration.

To address this evolving landscape, the integrated Network Enhanced Telemetry (iNET) program created the Telemetry Network Standards (TmNS), which are defined in an upcoming version of the Inter-Range Instrumentation Group (IRIG) 106 standards maintained by the Range

Commanders Council (RCC). IRIG 106 Chapter 23, Metadata Configuration, includes an Extensible Markup Language (XML) grammar that defines how telemetry devices are to configure themselves, the Metadata Description Language (MDL) [1]. This vendor-independent grammar is being increasingly adopted across the board by the telemetry industry. As a standard, its goal is to be capable of describing all telemetry capabilities for device configuration. However, because it can be used to set up a wide array of possible devices, it is very easy to describe a configuration that will not work for a given device. As such, a general method for creating MDL while considering the constraints (or capabilities) of individual devices is needed.

This paper introduces constraints and shows how they can be used to simplify flight test configuration by building correct-by-construction device configurations. Multiple configuration workflows are presented and the pros and cons of each are discussed. Finally, we describe an approach for rapid application development using constraints that can greatly reduce turnaround times when updating configuration tools for evolving changes in flight test instrumentation capability.

CONSTRAINTS

Constraints themselves are simply rules that control what something is allowed to do. Nearly everything one interacts with on a given day has constraints. A tree's branch can only bend so far before the wood snaps. A car can only turn so tight before the wheel locks. Some of these constraints, like the tree branch, can only be determined experimentally. Others, like the car, were designed by companies, and the exact values of the constraint were known before a steering wheel was ever manufactured.

In flight test instrumentation, the devices and sensors used are not the result of centuries of natural adaptation but were designed by teams of capable engineers. This means that, again, somewhere, the exact tolerances and constraints of these devices are known. There are some caveats to this statement. It is possible that some of a given device's internals were designed decades ago and the only record of what that device can really do is in an ancient compiler that can only run on a system unavailable to the device's current users. A more likely scenario is that the input language of a device has evolved over time, and what used to be a binary load custom built for the device is now a configuration file in a common language like MDL. To avoid rewriting the device's legacy logic there is an internal conversion to the legacy format from the MDL file, and the exact details of what happens after that conversion are inconveniently buried in a file cabinet which is not readily accessible.

In either case, in order for the device to be usable in today's world, it has to have a way to be configured. If we assume that the device is using a text-based language, where the grammar constructs in a file correspond to actions and settings on the device, then there are almost guaranteed to be constraints on that file because the text-based language can represent a broader range of parameters than the device can accept. The constraints may be as simple as "the maximum number of input channels is four", to more immediately useful ones such as "sample rates can only be powers of 2 from 64 to 8192 Hz", to particularly complex ones such as "if channel A or channel C are set to 8192 Hz, then channels B and D cannot be used". Because the

device configures based on the data in the file, it has to make sense of that data, and unless it can happily accept and do anything that is possible to express in the grammar, then there has to be some amount of constraints limiting what are valid files for the device.

There may also be constraints that exist outside of a device's sphere of influence, but still apply to input files fed to that device. Various levels of these constraints exist, and some terminology has already begun to develop to describe the most common. Vendor constraints come directly from the vendor, and directly describe capabilities of a device. System constraints encompass more than a specific device, describing the capabilities of any device that participates in a given system. These can include constraints such as "no device can send more than 25 Mb/s of data" or "the total of data sent from all devices cannot exceed 1 Gb/s". User constraints can apply at device levels or system levels, and are generally placed on a user by a higher organization to standardize or simplify device configuration, such as "in Acme Company, we will only use acquisition mode A". A system that manages constraints to create the configuration files must be able to synthesize and simultaneously take into account all these separate levels of constraints.

In legacy systems, these constraints tend to be buried in business rules and procedural code that must be used with the right incantation to get a configuration that is correct. It can be a grand undertaking to update these rules and screens that a user interacts with, where every given pixel has a hardcoded purpose, and any change causes cascading problems throughout the system because a checkbox that used to enable a feature is now on the other side of the terminal. But, even with all this, a file that makes it out of the system will be correct because the constraints that were so painfully crafted guarantee that the file will work.

In a modern constraints-based architecture, many of these pains go away. The logic of the constraints can be captured declaratively, stating what the rules must check, and not worry about how those checks are made. User interfaces can be designed independent of the validation and verification that must happen behind the scenes, through the power of separation of concerns. When a device's firmware updates and new capabilities are enabled (and old ones that used to work no longer do), it is just a matter of updating the declarative logic, and the system will again be guaranteed to produce correct files.

SIMPLIFYING CONFIGURATION WORKFLOW

The historical approach of embedding system knowledge in flight test configuration software, such as vendor tools or system-wide flight test configuration applications, necessitates code changes to the application's business rules when new devices need to be configured or system and user requirements change. Modifying these business rules is a software maintenance task that has proven to be costly and is likely to delay important feature updates making it to the user [2]. The TmNS attempted to address some of these issues by developing the TmNS configuration negotiation protocol that moves device constraints from the configuration application to the device. While this has been an important first step, it only addresses part of the problem and simply shifts the business rules from the application to the device. In this section, we will first examine the TmNS configuration negotiation protocol and then present a workflow

that shows how pulling the system knowledge out of the hardcoded business rules can address the shortcomings of negotiation and simplify the configuration workflow.

Negotiation Workflow

The TmNS configuration negotiation protocol is defined in IRIG 106 Chapter 25, Management Resources [3]. Negotiation is a sequence of steps executed between a configuration tool and a device to build a valid MDL instance document containing the device's configuration. The MDL instance document, or configuration file, resulting from negotiation will successfully configure the device because the device itself facilitated creation of the file.

Figure 1 shows the nominal workflow for creating an MDL file using negotiation. The user starts by entering configuration parameters in the configuration tool. Once the required configuration parameters have been entered, the user sends the MDL file to the device for validation and the device returns a pass/fail status indicating whether or not the MDL file contains a valid configuration for this device. If validation fails, an error message will also be returned to the user. To correct errors in the configuration, the user modifies the configuration parameters and repeats the validation step until it is successful. Once successful, the user requests the last validated MDL file from the device. The last validated file contains a valid configuration that has been annotated by the device with additional configuration parameters. The user can then use this last validated file to configure the device.

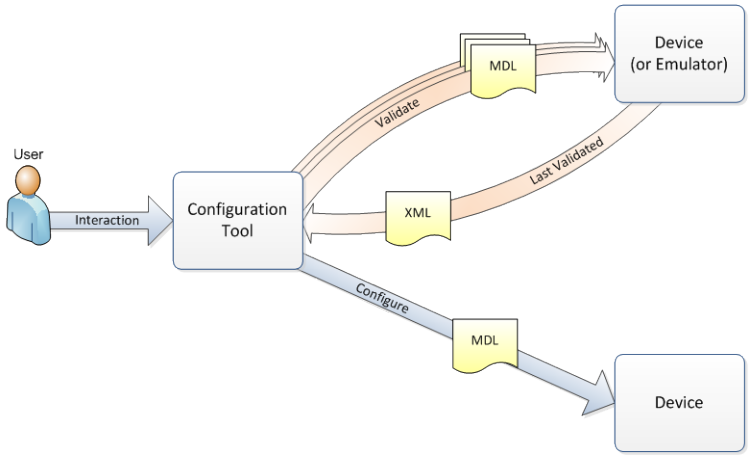

Figure 1. Negotiation Workflow

Negotiation is an improvement over the pure trial and error approach because the device feedback can guide user input to arrive at a valid configuration more quickly. While this workflow almost guarantees a valid device configuration, there are shortcomings that may already be evident. Most notably, a physical device (or device emulator) is required to be part of the workflow, and multiple iterations between the configuration tool and device may be required to reach a valid configuration. Additionally, negotiation cannot address system-level or additional user constraints.

Correct-By-Construction Workflow

A correct-by-construction workflow can be realized by pulling the system knowledge out of the hardcoded business rules and into a hierarchy of constraints [4]. That is, the constraints supplied at various levels are used to limit the possible user choices such that the resulting configuration will always work. As we saw with the negotiation workflow, configuration tools that guide user input will enhance usability and reduce the time needed to arrive at a valid configuration. With the correct-by-construction approach, the workflow can be further simplified by removing the physical device from the workflow and provide immediate feedback to the user as choices are made.

Figure 2 shows a workflow for creating a configuration that is correct-by-construction. The process starts with creating constraints files. Constraints can be created by various users and combined into a hierarchy that ultimately guides user input. Once the constraints are created, they are delivered to a developer and loaded into the configuration tool. The user enters configuration parameters and is guided by the tool to a correct-by-construction configuration for the device. Then, without the need for sending the configuration to the device (or simulator) for validation, the user can use this file directly to configure the device.

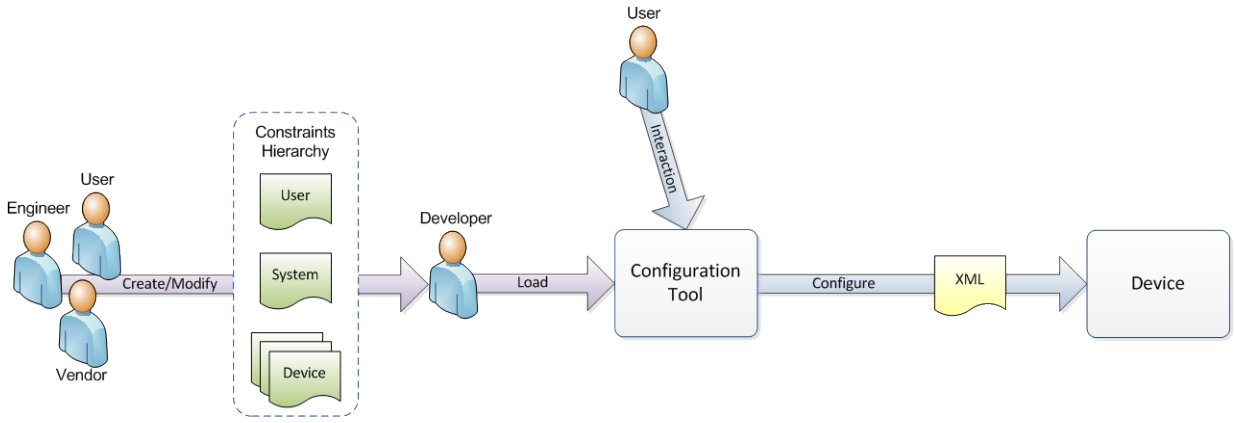

Figure 2. Correct-By-Construction Workflow

In some cases, it may not be practical to create constraints for all of the required configuration parameters. In this scenario, the correct-by-construction workflow may be augmented with vendor-supplied helper software (such as a compiler) to generate these final configuration parameters based on the user's other configuration selections. This approach differs from negotiation in that it is a one-time step that takes place after the configuration has been fully specified by the user under guidance of the constraints.

Figure 3 shows how the correct-by-construction workflow can be augmented with vendor-supplied helper software to add generated parameters to the final configuration. After the user enters the required configuration parameters in the configuration tool, they send the configuration to vendor-supplied helper software. The vendor-supplied helper software annotates the configuration with the necessary additional parameters and returns the configuration file. The user can now use this complete file to configure the device.

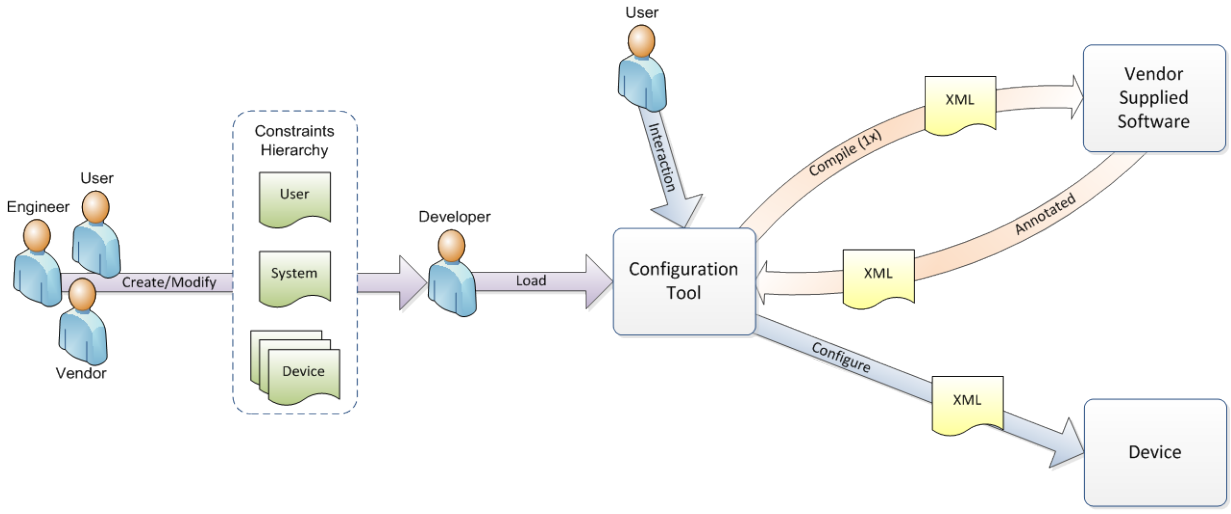

Figure 3. Correct-By-Construction with Vendor-Supplied Helper Software

The correct-by-construction approach has definite benefits over negotiation because of the immediate feedback presented to the user and the addition of system and user level constraints. However, the greatest benefit comes when comparing this workflow to the traditional approach of hardcoded business rules. By pulling the system knowledge out of code and into independent constraints files, software maintenance cost and schedule can be greatly reduced. This is because the configuration tools can be quickly and easily updated to support new devices, handle additional system restrictions, and keep up with changing user requirements by simply adding or modifying the appropriate constraints files.

AN APPROACH FOR RAPID APPLICATION DEVELOPMENT

This section describes a general approach for rapid application development using constraints that was developed by Southwest Research Institute (SwRI®).This approach has been successfully implemented by the Boeing Company in their Modular Instrumentation Setup Tool (MIST) [2] and is under consideration by the iNET program and other flight test organizations. XPath [5] was chosen as the constraints language due to its use in various domains related to XML, and specifically for its use in expressing constraints in other schema languages for XML such as Schematron and XML Schema 1.1. XPath is not a telemetry-specific language. XPath is the query language for selecting nodes from an XML document and is standardized by the World Wide Web Consortium (W3C). For MDL, XPath is already used to define uniqueness on fields and referential constraints checking that elements refer to the correct target, a common usage within schemas.

XForms [6], another XML-based technology made for gathering and processing XML data, was chosen over other available options for the reasons that it uses a declarative approach to constraints, and that there is a separation of the user interface component from the constraint logic. XForms provides a way for encapsulating XPath expressions and allows for the display of error messages and other visualizations that are needed by the user to make sense of the

validation results. The ultimate driving factor in the decision to carry the constraints in XForms is our unwillingness to fragment the MDL ecosystem by making changes directly to the MDL schema for each vendor and system. However, the iNET program is currently reviewing an MDL change proposal related to adding the XPath information directly into MDL instance documents in the future, which would allow for the knowledge to be transferred on a document by document basis without the interoperability concerns of system-specific changes to the MDL schema. In either case, it is the same XPath representation whether it is carried within the MDL instance document or in the XForms.

Constraints Examples

The following examples describe our general process of creating a constraint and provide a brief overview of the mechanics for creating and representing constraints in the XPath language in XForms. The process entails first determining which MDL element the constraint should apply to, writing the constraint using XPath expressions, and then providing user readable indication of constraints failure using XForms.

A sample constraint shown in Figure 4 will be used to walk through our technical approach. XPath works for any XML file, regardless of its schema, so we will use a simple XML example containing information describing a house in a neighborhood. This sample constraint captures the English-language sentence "The street number must be odd". By our above process, we first determine the element the constraint should apply to. Conveniently, our house has a <StreetNumber> element which can be directly used. This reference is located in the "ref" attribute. Then we need to write the XPath constraint which checks that the street number is odd, which is located in the "constraint" attribute.

```
<xf:bind ref="swri:StreetNumber" constraint="number(.) mod 2 eq 1"/>
```

Figure 4. XForms Constraint Example

Following the creation of the above constraint, we need to create feedback to the user in the case of validation failure. In XForms, an alert is used for this purpose. In this case, because we are checking that the street number is odd, the user will get the feedback that "Street numbers must be odd". This is shown in Figure 5.

```
<xf:input ref="swri:StreetNumber">
    <xf:label>Street Number: </xf:label>
    <xf:alert>Street numbers must be odd</xf:alert>
</xf:input>
```

Figure 5. XForms Error Message

Another constraint in the domain of MDL follows in Figure 6. The constraint encodes the English-language sentence "A SignalRange must have two ConditionParameter bounds whose values are not inverted". The constraint therefore checks that the lower bound (the `<ConditionParameter>` with a greater-than or greater-than-or-equal sign) is less than the upper bound (the `<ConditionParameter>` with a less-than or less-than-or-equal sign).

```
<xf:bind ref="mdl:SignalRange"
constraint="
count(mdl:ConditionParameter) ge 2
    and
    number(mdl:ConditionParameter[mdl:ConditionOperation = ('&gt;=', '&gt;')]/mdl:ConditionValueFloat)
        lt
    number(mdl:ConditionParameter[mdl:ConditionOperation = ('&lt;=', '&lt;')]/mdl:ConditionValueFloat)">
```

Figure 6. Example MDL Constraint

Following the creation of the above constraint, we need to create feedback to the user in the case of validation failure. In this case, because we are checking that the signal range must have two `<ConditionParameter>` bounds whose values are not inverted, the user will get the feedback that "A SignalRange must have 2 ConditionParameter bounds whose values are not inverted". This is shown in Figure 7.

```
<xf:group ref="mdl:SignalRange">
    <xf:label>SignalRange</xf:label>
    <xf:alert>A SignalRange must have 2 ConditionParameter bounds whose values are not inverted</xf:alert>
</xf:group>
```

Figure 7. XForm Error Message for MDL Example

Rapid Application Development

After the constraints have been created, the next task is building a user interface that can actually use them. We already have the pieces that are relevant to the user, through their creation with the constraints, which gives us several options. One option is to autogenerate the relevant user interface straight from the schema, and add a layer of customization on top that controls and directs what the user can see (skinning the UI). Another is to handpick a small selection of elements and build them straight into the skin itself. Varying tradeoffs can be made when weighing extensibility or performance.

In any case, there are more opportunities for aiding the user in system operation. Templates, essentially sections of a configuration with some values pre-populated, can be created and integrated for situations where much of the correct parameters are already known. For example, a user wants to create a specific measurement, but due to the current channel settings, the sample rate and signal range have only one acceptable value. The user can set up the measurement name and precision, so some input is required, but the sample rate and signal range can be pre-set at the time the measurement is added.

The important part of a system built this way is the ease of updating the constraints. If a device's capabilities change through a release, the user interface is divorced from the logic that needs to change. At most, a new feature is added and new fields will need to be created or exposed. The declarative logic will have to change, and if a template assumed values that are no longer true, the template will need to change. But there is no dramatic rework of pixel locations and purposes, no drastic rewrite of all logic in the system because a basic assumption is suddenly wrong. Just plug the new constraints in, and let the automatic validation from the XForms tell the users what new configurations are now accepted by the device.

CONCLUSION

Every device has constraints, whether it is the most feature complete device available and usable for every flight test need, or it is built on decades-old but proven methods. The configuration parameters have to be set by a user, and they have to be conveyed to the device, and this transfer is the ideal location to place computable constraints to insulate the device from direct contact with the user, only allowing files that are correct to be seen by the device. Constraints will guide the user's interactions to create a valid file, giving them information about what is and is not allowed at every step. The output file, through a correct-by-construction process, is guaranteed to be accepted by the end device.

When requirements and capabilities change, the modularity and separability of declarative constraints make them significantly easier to update than deeply-coupled business rules. Turnaround times on a system update and associated regression testing can go from weeks and months to only days or less, and again the system will be guaranteed to make correct files for the target devices. User interfaces can evolve and progress separately from the logic that drives their validation. Even schema changes on the underlying configuration language do not have to cause dramatic code restructuring.

Flight test configuration systems do not have to be unwieldy applications with hardcoded business rules that limit the responsiveness needed for the continually changing requirements of the flight test environment. The constraints driven approach presented in this paper provides a proven path to simplifying the development and maintenance of configuration systems for the increasingly complex flight test systems.

REFERENCES

[1] *Metadata Configuration*, IRIG 106 Chapter 23 "Pink Sheet", August 2016.

[2] Neumann, M., Moore, J., Pantham, S., Moodie, M., Noonan, P., Whittington, A.: An Adaptable Constraints-based Metadata Description Language (MDL) System for Flight Test Instrumentation Configuration, Proceedings of the European Telemetry and Test Conference, Nuremberg, Germany, May 2016.

[3] *Management Resources*, IRIG 106 Chapter 25 "Pink Sheet", August 2016.

[4] Jackson E., Sztipanovits J.: Correct-ed through Construction: A Model-based Approach to Embedded Systems Reality, 13th Annual IEEE International Conference on the Engineering of Computer Based Systems, CD-ROM, Potsdam, Germany, March 27, 2006.

[5] https://www.w3.org/TR/xpath/

[6] https://www.w3.org/TR/xforms/