Formal Data Validation : Formal Techniques Applied to Verification of Data Properties

Mathieu Clabaut, Christophe Metayer, Éric Morand

Formal Data Validation

Formal Techniques Applied to Verification of Data Properties

Mathieu Clabaut[1], Christophe Metayer[2], and Éric Morand[3]

[1] Systerel, Aix-en-Provence, France mathieu.clabaut@systerel.fr
[2] Systerel christophe.metayer@systerel.fr
[3] CNES, Toulouse, France eric.morand@cnes.fr

Abstract. Nowadays lot of critical systems used in aeronautic, aerospace, automotive and railway domains are designed to be highly configurable. In such systems, the data configuration validity is of tremendous importance. The railway domain does already apply some formal techniques to validate configuration data. This study, funded by the CNES, demonstrates the suitability of such methods for validation of flight software configuration data.

Topic. Dependability, safety, security, quality of service
Domain. Verification and validation

1 Context and Goals

1.1 Context

Complex systems in the aerospace domains are required to be more and more adaptable and reusable, and hence tend to be based on generic software configured by means of data parameters.

Validation of such generic software follows two approaches: either the generic software and its configuration are validated together for the targeted deployment or the software and its configuration are validated separately if possible or else in a two stage validation where data are validated before the configured software.

Our experience about data validation is that:

- it is often done by procedural means, an ad-hoc software being developed to achieve the verification.
- properties on configuration data, whenever identified are often considered as second class citizens.

It has been shown that such verification tools are difficult to maintain when the properties evolve.

The algorithmic implementation of the properties not only is being written by someone who is not acquainted to the domain they are related to (namely a software designer) but also has the unwanted side effect of obfuscating them. An external assessor thus has the greatest difficulties to ensure that desired properties are those verified by the tool.

Based on some successful experiments on applying formal techniques to validation of configuration data for the railway domain, we were willing to apply them to space flight software, and especially to basic software (BSW) configurations.

1.2 Goals and Planning

Expected benefits of the study are:

- clear property identification and formalisation;
- automatic property verification on binary code;
- demonstration of feasibility, usability and improvements over previous verifications.

Two main tasks were devised when planning the study:

1. harvest the known and possibly unknown properties of BSW data configuration and formalize them within a rather simple mathematical language (first order predicate, set theory).
2. write some tools to automatically extract data from binary code and verify the properties on these data.

1.3 Data Validation for Railways

A bit of history Historically, interlocking systems were designed by means of physical levers and cotter pins and were carefully designed to avoid enabling conflicting routes. A century ago, all-electric interlocking made up of electrical relays began to be designed. However each interlocking system is still specifically designed to avoid enabling conflicting routes on a well defined area.

Since the late 1980s, most interlocking engines are fully implemented in software. As the cost for developing such a SIL4 system is quite high, manufacturers are faced with the problem of designing a *generic enough* interlocking engine[4] which would accommodate most of the existing ground configurations by means of configuration.

Such a (big) configuration data set is then a full contributor to the global safety level, and as such must be validated thoroughly.

[4] Note that other subsystems of *automatic train protection* systems have the same requirements upon safe and voluminous configurations.

Data Validation, the old way Validating huge data configuration set against complex properties is quite a daunting task, and needs automation. For this purpose, manufacturers began to develop software specifically designed to verify the configuration data set against some expected properties.

Those software were written in C or C++ and the expected properties as well as the format of the configuration data were hard-coded in their internals. The following consequences soon arose:

validation cost: the whole *data validation software* has to be validated, not only the properties,

actors involved: the domain experts have no means to check that the properties verified by the software are the expected ones and conversely, the software engineers aren't domain specialists. This sometime leads to inconsistency between what was checked and what ought to be checked,

evolution cost: modification of a property requires to modify the software and to revalidate it,

product line: developing a similar subsystem requires to develop again most of the *data validation software*

The whole process was cumbersome, error prone and expensive.

Data Validation, the new way Some years ago, RATP began to rethink the way configuration data were validated and a tool was developed upon the opensource platform Rodin [2] to validate those data against safety properties which would be formally specified.

The chosen formal language is a subset of the B language [4] which can simply be seen as a first order predicate logic over set theory using a notation which shares a lot with standard mathematical notation ($\exists, \forall, \vee, \wedge, \cup, \ldots$) which was perfected for several centuries and is well understood by most engineers and which, due to its formal nature, is also understandable by computers (see figure 1).

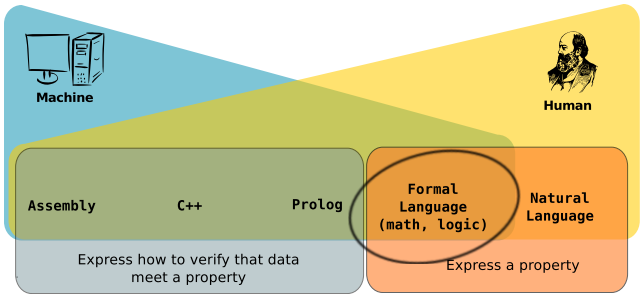

Fig. 1. Mathematical formal language as the best compromise between human and computer

The foundations were then laid for the building of a modular software platform allowing verification of formal properties over a wide range of data.

The following sections will elaborate a bit on the case study perimeter, properties and data considered, tools developed and used and the results obtained.

2 Case Study

We describe hereafter the chosen case study which consists in the configuration of the GPMU BSW.

2.1 Satellite GPMU

The Generic Payload Management Unit (GPMU) is the part of the satellite (hardware and software) in charge of hosting the payload software which will provide the set of functions enabling the satellite to fulfill its mission (by contrast with the platform part, which provides basic services like energy sources and positioning system).

The GPMU architecture is described in figure 2, and is based on the XtratuM hypervisor [3].

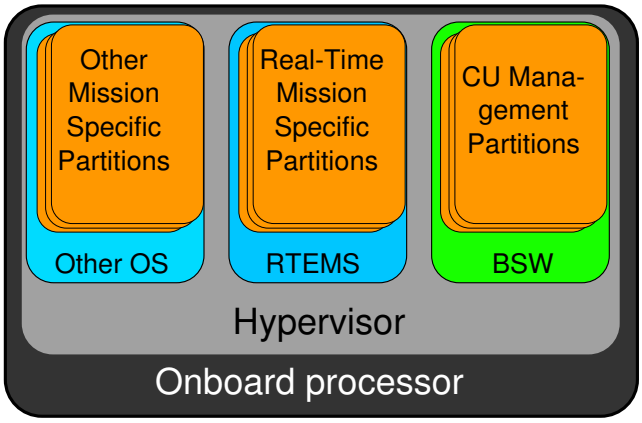

Fig. 2. GPMU architecture

GPMU is composed of a resident software (not shown) which loads and launches the hypervisor which offers a spatial and temporal partitioned environment for running different OS. The hypervisor virtualises services offered by the physical platform in order to make them available to the various hosted OS.

The Basic Software (BSW) is a subsystem which isolates the payload management functions from the mission specific functions. It is composed of several distinct partitions which offer functions related to:

- mode management,
- inter-partition I/O management,
- non volatile memory management,
- hardware and instrument events management, …

We will focus our analysis on the MMDL BSW partition which offers high level services to manage application partitions (memory mapping, partition monitoring,…), and on the IOSERVER BSW partition which offers services to manage inter-partition and device communications. Those services are provided by mean of the PMU protocol and all partitions, whether they belong to the Basic Software or not, will use the PMU protocol for inter-partitions communications with the Basic Software. Others communications, for instance between two non BSW partitions, can be opaque and so may not use the PMU protocol.

Partitions outside BSW offer mission specific functionalities.

2.2 Partitions Configuration

While each development team is responsible for its partition development, the person responsible for integration shall produce the hypervisor configuration table by fulfilling the following tasks:

- collect characteristics and constraints for each partition,
- configure the hypervisor with each partition (memory mapping, …),
- configure the I/O management for each partition,
- configure the inter-partition I/O ports,
- configure the I/O scheduler,
- configure the partition scheduler,
- test and validate the global GPMU behaviour.

Our study will concentrate on the production of the BSW MMDL and IOSERVER configuration tables, whose complexity comes from the number of dependencies between the a priori chosen partition scheme, and the a posteriori constraints (I/O, memory mapping, …).

3 Properties

3.1 What does *Data Validation* mean?

And what are data anyway? The set of considered data are *constants* that contribute to the configuration of a system. They may be

- physical constants,
- parametric constants which are used to define a command law,
- mission specific constants (which may be uploaded),
- parametric constants specific to the design choices,
- validation and test data,
- constant produced while defining the configuration data (authors, creation date, version, integrity,…)

Properties link different types of data. They exist independently from the system being active or not and define *invariants* over the system.

Validating data is the act of verifying that the configuration data verify the properties (most often, safety ones) identified.

Of course, validating data is necessary but not sufficient to ensure the correctness of the system behavior.

3.2 Property Identification

The first part of the job is to *informally* identify which properties shall be verified by the configuration data. The following available documentation was taken into account:

- XtratuM hypervisor User Manual,
- XtratuM hypervisor Reference Manual,
- BSW User Manual,
- A first identification of some coherence rules.

We began to dig out and express in natural language some required properties, which may be categorized as follow:

A properties relative to the domain of scalar constants.
One shall mainly verify that a scalar parameter is defined within its bounds or that its extremum values are compatible with the hypervisor configuration,

B equality constraints.
For example, verify that the size of a structure is equal to the value of another parameter,

C parameter typing.
Verify that all data loaded in a structure respect the structure type,

D coherence between data.
For example, check that some memory spaces are disjoints for a given partition, or that the communication port characteristics for interconnected ports are compatibles, or check unicity within specified structures…

The configuration scheme chosen for the BSW which compiles configuration files written with the help of the C language, ensures that properties of type **C** are verified. Those properties won't then be addressed in the following.

An example of simple property of type **A**, extracted from the available documentation is shown hereafter in figures 3 and 4. The actual configuration as found in the `.c` file is shown in figure 5.

Parameter	*Description*	*Value*
MIN_MEMORY_OPERATION_PORT_SIZE	MIN size of memory request ports (bytes)	48
MAX_MEMORY_OPERATION_PORT_SIZE	MAX size of memory request ports (bytes)	256

Fig. 3. Constraints on memory port size bounds as defined in the BSW manual

Parameter	*Description*	*Min*	*Max*
MMDL_Memory_In_Port_Size	Size of the port used to received memory operation requests (bytes)	48	256
MMDL_Memory_Out_Port_Size	Size of the port used to send memory operation results (bytes)	48	256

Fig. 4. Constraints on memory port size as defined in the BSW manual

```
UInt16 sMmdlMemOpInPortSize = 128;
UInt16 sMmdlMemOpOutPortSize = 128;
```

Fig. 5. Memory port size as defined in the configuration file (C language)

A more complex property of type **C** is inferred from our understanding of the system and from actual data configuration as shown in figure 6. It says that memory mapping defined for a given partition id are disjoints.

```
MmdlSMemoryArea tMmdlMemoryMappingTable[14] =
{
   {0x34000, 0x17000    , 1, PARTITION_CODE, FLASH},
   {0x4B000, 0x800      , 1, CONFIG_TABLE, FLASH},
   {0x4B800, 0x200      , 1, CUSTOM_TABLE, FLASH},
   {0x4C000, 0xC000     , 2, PARTITION_CODE, FLASH},
   {0x4B000, 0x800      , 2, CONFIG_TABLE, FLASH},
   {0x5C000, 0x10000    , 3, PARTITION_CODE, FLASH},
   {0x40070000, 0x1C0000, 1, PARTITION_CODE, RAM},
   {0x40230000, 0x20000 , 1, CONFIG_TABLE, RAM},
   {0x40250000, 0x20000 , 1, CUSTOM_TABLE, RAM},
   {0x40270000, 0xC000  , 2, PARTITION_CODE, RAM},
   {0x4027C000, 0x4000  , 2, CONFIG_TABLE, RAM},
   {0x40280000, 0x10000 , 3, PARTITION_CODE, RAM},
   {0x40260000, 0x400   , 2, CUSTOM_TABLE, RAM},
   {0xA0000, 0x100      , 1, NONE, FLASH}
};
```

Fig. 6. Memory mapping table

Some properties may also come from standard conformance or from other design constraints, like saying that existing hardware should be used when possible. One example of such a property is: *UART components have normalized speeds*.

However it is often very difficult to exhaustively identify properties that should hold upon configuration data once system design is over.

This last statement promotes an early integration of property management in a system development cycle.

3.3 Property Formalisation

Once the properties are identified, we define a formal model which describes the data constants and properties as axioms over the constants. The use of event-B [1] allows us to decompose the model as shown hereafter.

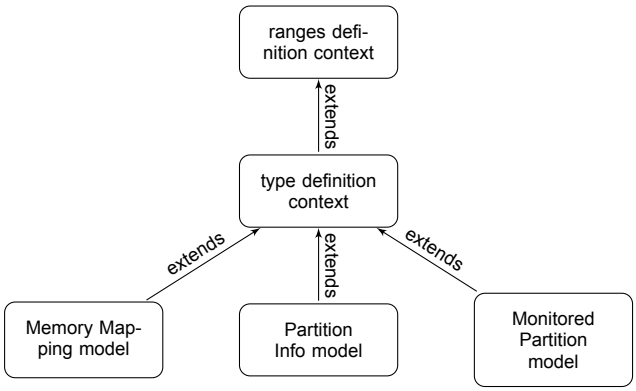

Fig. 7. Constant and property model architecture for MMDL partition

Two models were defined, one for the `MMDL` partition which defines the properties about the *memory mapping table,* the *partition information table* and the *monitored partitions table*; and one for the `IOSERVER` partition which defines the properties about the *device description table* and the *scheduling plan table*.

[5] where `A → B` is the set of total function from A to B.

```
   // Range definition for memory operation port size
@axm1 : MIN_MEMORY_OPERATION_PORT_SIZE ∈ ℕ
@axm2 : MAX_MEMORY_OPERATION_PORT_SIZE ∈ ℕ

   // In and Out port size definition
@axm1 : sMmdlMemOpInPortSize  ∈ ℕ
@axm2 : sMmdlMemOpOutPortSize ∈ ℕ
```

```
   // Actual property over the port size being in its
   range
@axm11: sMmdlMemOpInPortSize ∈
    MIN_MEMORY_OPERATION_PORT_SIZE
     .. MAX_MEMORY_OPERATION_PORT_SIZE
@axm12: sMmdlMemOpOutPortSize ∈
    MIN_MEMORY_OPERATION_PORT_SIZE
     .. MAX_MEMORY_OPERATION_PORT_SIZE
```

The constraint about disjoint memory mapping being defined for a given partition id may be formalized as follow[5]:

```
   // MmdlSMemoryArea fields definition
@axm4 : StartAddress ∈ 1 .. MemoryMappingTableSize → ℕ
@axm5 : Length ∈ 1 .. MemoryMappingTableSize → ℕ
@axm6 : PartitionID ∈ 1 .. MemoryMappingTableSize → ℕ
```

```
   // property for disjoint memory mapping
@axm9 : ∀ i,j ·  i ∈ dom(StartAddress)
  ∧ j ∈ dom(StartAddress)
  ∧ i ∈ dom(Length) ∧ j ∈ dom(Length)
  ∧ i ∈ dom(PartitionID) ∧ j ∈ dom(PartitionID)
  ∧ i ≠ j
  ∧ PartitionID(i) = PartitionID(j)

  ⇒ StartAddress(i) .. StartAddress(i)+Length(i) − 1
  ∩ StartAddress(j) .. StartAddress(j)+Length(j) − 1
  = ∅
```

Where `dom(f)` denotes the domain of function `f`.

The constraint about UART speeds may be formalized as follow:

```
   // Constant definition and typing
@axm2 : partition(COMPONENT_TYPE,
                  {UART}, {I2C}, {OSlink})

@axm7 : Speed ∈ 1 .. DeviceDescriptionTableSize → ℕ

@axm8 : ComponentType ∈
  1 .. DeviceDescriptionTableSize → COMPONENT_TYPE
```

```
   // Actual property
@axm13 : ∀y · y ∈ dom(Speed)
   ∧ y ∈ dom(ComponentType)
   ∧ ComponentType(y) = UART
   ⇒ ran(Speed) =
     {110,300,1200,2400,
      4800,9600,19200,38400,
       57600,115200,230400}
```

Where `ran(f)` denotes the range of function `f`.

Beyond the possibility of data validation, having defined such a formal model for properties allows:

- a formal expression not only for the properties but also for the associated data structures,
- a proof of well-definedness of properties and a verification of their types,
- the use of a *simple* and common enough formalism, shared by the different actors,
- a separate validation for the proof mechanism and for the data (the proof mechanism shall be validated only once),

4 Data Extraction

4.1 General Case

From our previous experiences in validating data for the railway domain, it appears that data may come from various sources and be encoded into various formats (DBMS, XML files, structured text files, flat text files, ADA or C source code). It is not conceivable to devise a piece of software which would be able to manage such a variety of sources and format.

Thus the software that verifies formal properties on data (lets call it *data prover*) was designed to be data source agnostic but expandable by means of plug-ins to be able to get data from sources that were not envisioned at the time of initial design.

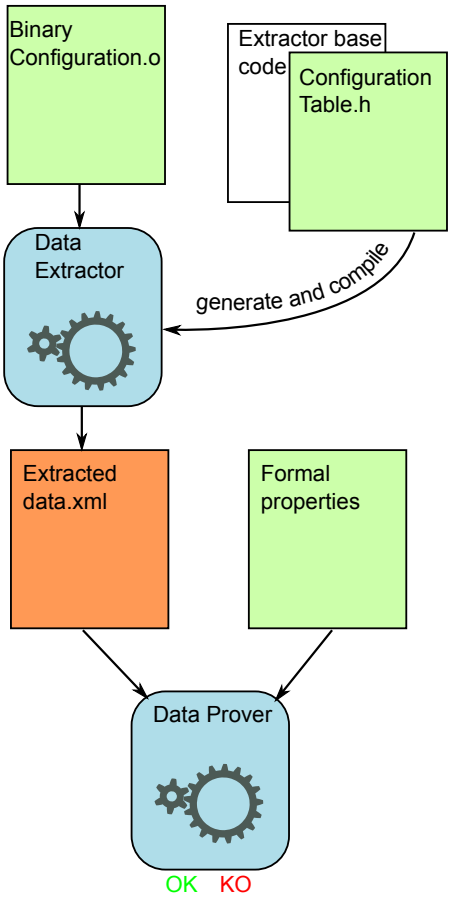

Fig. 8. Data extraction and verification process

4.2 GPMU Case

The configuration data for the GPMU are built by mean of `.c` source files compiled into binary objects and linked within the XtratuM platform.

With safety in mind, we want to check the data as late as possible in the configuration process in order to prevent as much as possible data corruption to happen after the verification being done.

The very last place where most of the configuration data may still be understandable for an external piece of software, is in the ELF binary object produced by compiling the configuration `.c` source files. However data types may not be inferred from the binary ELF file and shall be provided to the extractor by another channel.

With the objective to be as efficient as possible, we have chosen to develop an extractor with buit-in data type information. It is a bit less convenient as the extractor shall be compiled again when the data types change, but it allows us to exhibit some results in a limited time and budget.

The overall process is shown in figure 8 hereafter.

5 Tools

A short overview of the tools developed or used during this study is given is the following section. The tools are:

- a data extraction tool for harvesting configuration data from binary objects,
- a formal development platform for checking that properties were well typed and well defined (i.e. meaningful),
- the core of the *data prover* tool, which was improved and used over the properties and associated configuration data.

5.1 Data extraction from Binary Objects

A tool was coded in C, to extract the data from the binary objects based on their ELF format, which by mean of relocation tables allows us to find values of constants defined as `static const` in the `.c` piece of code which defines the configuration data.

In order to simplify the whole development, the expected data types, which can not be inferred from the data provided by the ELF format, are hard-coded in the extractor tool which produces a simple XML file. This XML file will then be used by a small *data prover* plug-in to translate the XML representation into the targeted internal data constant representation.

An extract of such an XML file obtained from actual analysis of binary configuration objects is shown in figure 9.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<constants>
<scalar name="lOutDataRxQueuingPortsNumber" type="UInt32
    " value="3"/>

<scalar name="lInDataTxQueuingPortsNumber" type="UInt32"
     value="3"/>

<scalar name="lOutDataAckQueuingPortsNumber" type="
    UInt32" value="3"/>

<array name="cTableOutDataRxPortNames" type="Int8*">
   <string value="QIOPFTCO"/>
   <string value="QIOI1TMO"/>
   <string value="QIOI2TMO"/>
</array>
<array2 name="cInDataTxConnectionTable" type="UInt8">
   <row>
      <number value="1"/>
      <number value="0"/>
      <number value="0"/>
   </row>
   <row>
      <number value="0"/>
      <number value="1"/>
      <number value="0"/>
   </row>
   <row>
      <number value="0"/>
      <number value="0"/>
      <number value="1"/>
   </row>
</array2>
</constants>
```

Fig. 9. Intermediate XML data description extracted from binary objects (short extract)

5.2 Static Checking of Formal Properties

We choose to use the existing Rodin [2] platform as supporting tool to develop the formal model of properties. It brings the following advantages:

- easy input of mathematical formulas,
- easy formula reading by mean of UTF8 characters and syntax highlighting,
- use of event-B [1] model structures to decompose the formal model,
- transparent type checking of the formal model,
- transparent proof obligation generation for well-definedness,
- transparent automatic proving attempt of those proof obligations,
- assisted manual prover for the remaining undischarged proof obligations if any,
- easy model import by the *data prover*.

Thus, while any data verification tool developed in a low level language (say C, for example) would have allowed to check a string pointer against an integer without any hiccup, it won't even be possible to type check the corresponding property in our formal model.

This definitely leads to a data validation of better quality.

5.3 Data Validation against Formal Properties

The last tool developed within this study is the core *data prover*, based on existing pieces of code from the Rodin platform, which once fed with:

- the formal context defined as event-B contexts components,
- the properties to be verified defined as event-B axioms in terms of constant and types defined in the associated contexts,
- the data XML file produced from the binary configuration objects by the data extractor tool,

will verify in turn each property (axiom) against the provided data and will produce a status similar to the one provided in figure 10.

```
OK axm1: lOutDataRxQueuingPortsNumber ∈ ℕ
OK axm2: lOutDataRxQueuingPortsNumber ≤
    OUT_DATA_RX_QUEUING_PORTS_MAX_NUMBER
OK axm3: lInDataTxQueuingPortsNumber ∈ ℕ
OK axm4: lInDataTxQueuingPortsNumber ≤
    IN_DATA_TX_QUEUING_PORTS_MAX_NUMBER
OK axm5: lOutDataAckQueuingPortsNumber ∈ ℕ
OK axm6: lOutDataAckQueuingPortsNumber ≤
    OUT_DATA_ACK_QUEUING_PORTS_MAX_NUMBER
OK axm7: cTableOutDataRxPortNames ∈  0 ‥ (
    lOutDataRxQueuingPortsNumber − 1) ⇸ ℕ
OK axm8: cTableInDataTxPortNames ∈  0 ‥ (
    lInDataTxQueuingPortsNumber − 1) ⇸ ℕ
OK axm9: cTableOutAckPortNames ∈  0 ‥ (
    lOutDataAckQueuingPortsNumber − 1) ⇸ ℕ
OK axm10: ran(cTableOutDataRxPortNames) ∩ ran(
    cTableInDataTxPortNames) ∩ ran(cTableOutAckPortNames)
    = ∅
OK axm11: lTableOutDataRxPortSizes ∈ 0 ‥ (
    lOutDataRxQueuingPortsNumber − 1) → ℕ
OK axm12: ∀y ·y ∈ ran(lTableOutDataRxPortSizes) ⇒ y ∈
    OUT_DATA_RX_PORTi_MIN_MSG_SIZE ‥
    OUT_DATA_RX_PORTi_MAX_MSG_SIZE ∧ y mod 4=0
OK axm13: lTableInDataTxPortSizes ∈ 0 ‥ (
    lInDataTxQueuingPortsNumber − 1) → ℕ
OK axm14: ∀y ·y ∈ ran(lTableInDataTxPortSizes) ⇒ y ∈
    IN_DATA_TX_PORTi_MIN_MSG_SIZE ‥
    IN_DATA_TX_PORTi_MAX_MSG_SIZE ∧ y mod 4=0
OK axm15: lIoDevicesNumber ∈ ℕ
KO axm16: lIoDevicesNumber ≤ IO_DEVICES_MAX_NUMBER
OK axm17: partition(CONNECTION_STATUS,{ON},{OFF})
KO axm18: cInDataTxConnectionTable ∈ 0 ‥ (
    lInDataTxQueuingPortsNumber − 1) × 0 ‥ (
    lIoDevicesNumber − 1) → CONNECTION_STATUS
OK axm19: ∀·yy ∈  0 ‥ (lInDataTxQueuingPortsNumber − 1)
    ⇒((({y} × ℕ) ⩤ cInDataTxConnectionTable ⩥ {ON})~ ∈
    CONNECTION_STATUS ⇸ ℕ × ℕ
KO axm20: cOutDataRxConnectionTable ∈ 0 ‥ (
    lOutDataRxQueuingPortsNumber − 1) × 0 ‥ (
    lIoDevicesNumber − 1) → CONNECTION_STATUS
```

Fig. 10. Verification results (where some properties are falsified)

6 Results

6.1 Achieved Work

This study allowed us to show that the validation of data based on formal property expression, which has already been used in the railway domain is also applicable to on-board basic software configuration for the space domain.

Even if the properties verified in this study are most often simple ones, our past experience has shown that such a process scales well to more complex properties

upon a lot of data provided that associated data are *easy enough* to extract[6].

It was also shown that a difficult task is to identify the properties to be checked. They are not always written down in available documentation. Having to formalize the properties won't help much in finding them, but it is a great mean to capitalize those properties and to integrate them in the whole development cycle.

6.2 Further work

Domain of Application It would be interesting to extend the data verification for on-board software to complete satellite system (including the satellite platform, with its related configuration and calibration constraints).

There are also probably lots of other domains outside railways and space that may benefit of such an approach, provided people are conscious of the existence and the role of the properties that should hold upon their configuration data.

Tool Improvements Tools developed within this study may be improved in several ways:

- data extraction should not need recompilation upon data type change,
- natural language expression of properties shall be integrated into the formal model or a document shall be produced mixing formal and informal expression of properties to allow validation by domain experts.
  The report result should also refer to the natural language expression of properties,
- a user interface should be provided to allow inspection of the data set that falsifies a property, to ease the spotting of the configuration data responsible for the property falsification.

Get More from Formal Methods We could imagine to integrate completely the verification framework within the Rodin platform, by making the property verification tool being one of the available *prover*. It would allow verifiers to decompose and prove very complex properties with the help of the whole tool set without restricting to symbolic evaluation.

Identifying Properties Afterward is probably one of the most difficult point raised by this study. Having the tools and methods for deploying easy data validation would probably be a good incentive for including property management since the beginning of the system development cycle.

References

[1] Event-b — a formal method for system-level modelling and analysis. `http://www.event-b.org/`.

[2] Rodin — rigorous open development environment for complex systems. `http://rodin.cs.ncl.ac.uk/`.

[3] Xtratum. `http://www.xtratum.org/`.

[4] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.

Glossary

BSW Basic Software

ELF Executable and Linkable Format

GPMU Generic Payload Management Unit

interlocking system Railway system which establishes train routes by mean of points and signals.

MMDL Mode Management and Data Load

SIL *Safety Integrity Level* a relative level of risk-reduction provided by a safety function.

PMU Payload Management Unit

[6] For example, our solution won't help for proving properties over unstructured data to be extracted from PDF documents.