Formal proofs applied to system models

Évelyne Contejean, Andrei Samokish

Formal proofs applied to system models.

Évelyne Contejean, Andrei Samokish

LMF — Université Paris-Saclay, CNRS, ENS Paris-Saclay

Abstract

Usually, the description of nuclear equipment by the FMEA (Failure Mode and Effects Analysis) method can be of considerable length (up to 5,000 lines); on the other hand, the number of rules used for the verification of this equipment is small. In addition, upstream, there is the question of trust in the tools that generate these descriptions for complex equipment, that is to say, made up of several thousand objects (requirements, functions, interfaces, behaviors).

Our objective is to formally prove in the Coq proof assistant the accuracy and exhaustiveness of the safety analysis for these nuclear equipments, by proposing a more modular and more abstract framework which is based on an axiomatic description of the systems. - these axioms being either accepted or proven independently of the verification itself. The very nature of the inductive types of Coq is perfectly suited to the description of complex systems, which are systems of systems in interaction. We have a first prototype that demonstrates the feasibility of this approach, for simple properties which are effectively proven by reflection - for example, the study of flows, which must first be produced and then consumed exactly once in the whole system.

We open the way to a general approach for the proofs of the properties of a (model of) systems and study the kind of properties that can be demonstrated in this way. Finally, we propose the approach for faithful translation of a DSL (domain-specific language) tool into Coq and the proof of its correctness.

1 Motivation

For about 15 years, the idea that systems can be modeled has been developing, and more and more processes, including safety and security analysis, have started to be backed by models. Such models can be used to perform simulations but may also reduce complexity and turn complex problems into smaller ones from a logical point of view. This is the case for Failure Mode and Effect Analysis (FMEA), where the risk analysis of a complex system is turned into an extensive list of risk reduction means for each couple of functions and components in the system. More generally, for complex systems, the safety case becomes a complicated document or set of information that transforms a high-level goal into a huge number of much simpler goals. An FMEA for a Nuclear Steam Generator (used as an example in this paper) is in the order of magnitude of thousands of lines.

The idea that software tools can help handle this complexity has been in the air for a long time. However, software tools are complex systems that can be challenged as well as any other complex software. When dealing with complex system models, the question of how confident we can be in the tools to handle all situations and faithfully describe the system of interest is at stake. In the software domain, Formal Methods (FM) are already used to verify that complex software performs correctly (see proof of C compiler by X. Leroy [10]), so it seems interesting

to investigate a similar approach for systems models and related tools.

We notice that the general definition of a system is inductive: broadly speaking, systems are made of (sub)systems in interactions, so when trying to prove some property on a system, it is logical to deduce that property from some properties of its subsystems. More generally, systems engineering is mainly built upon inductive breakdown structures: Relation Block Diagram (RBS), Functional Block Diagram (FBS), Project breakdown structure (PBS)… Classical safety methods, although inductive or deductive, are also based on the inductive design of systems. For this reason, we decided to investigate inductive types as a foundation for the formal definition of a system.

With this objective in mind, we should take care of two critical issues:

First, FMs are usually applied to detailed system behavior. Using FM methods seems very useful when the logic of the system is already defined. For example, formal Event-B specifications applied to FMEA give interesting results [5]. However, the later a problem is discovered, the more expensive the solution [11], so it is relevant to investigate the use of FM at the specification level, especially when high-level performance or non-functional requirements are refined into lower-level requirements. The development of Model-Based Systems Engineering provides an opportunity here because these tools and methods keep track of the refinement of the system design from the external analysis to the detailed component description. This objective is upstream w.r.t. the usual application of FM

Second, there can be significant drawbacks in FM application (a detailed description of problems can be found in [4]:

- FMs are difficult to learn and apply. Many assurance practitioners perceive the usage of FMs as negative.
- The number of practical (comparative) case studies using preliminary initial system design FMs is still too low to draw valuable and firm conclusions on FM effectiveness.
- FMs have been suffering from fragile effectiveness and productivity in the context where some physical processes are supposed to be controlled with a computer system.
- The effectiveness of formal models can be significantly reduced because of uncontrollable gaps between models and their implementations.

So, we need to provide an intermediate step between the FM-proof assistant and the system and safety model. The complexity of FM needs to be hidden from the end-user, and at the same time, it shall apply efficiently. For this reason, we propose a DSL (domain-specific language) as a front-end user interface that will be accessible to the engineer without knowledge of FM.

Our approach intends to make a bridge between system model description and formal systems, simplifying the dialog between engineers and mathematicians and allowing them to investigate system models with FM in the initial design stages.

2 Process and vocabulary

Fig. 1. Explains the different transformations that allow translating a system model faithfully into a Proof assistant program. In this section, we introduce both the process and the vocabulary corresponding to each translation step.

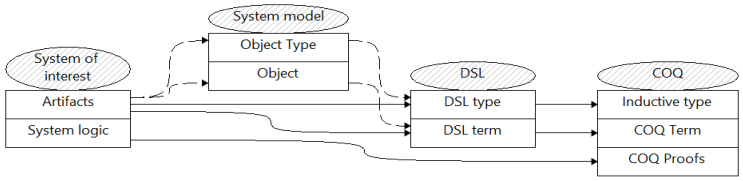

Fig. 1. Model formalization process

1. At the very beginning, Systems engineers describe systems through *System item*, *system interactions* (flows), *system artifacts* (used to describe, specify, and communicate about the system) (see INCOSE handbook [13]) - *artifacts* includes requirements, functions, flows, components, systems (in the meaning combination of functions and components). For systems, we may differentiate artifact categories or types from artifact instances. An artifact category may be 'function' while an artifact instance of "function" would be 'function for heat transfer between primary and secondary circuit'.
2. Modeling the System of Interest is getting a standard way for specifying.
   During model formalization, engineers describe system *Artifacts* representing parts of the system of interest. Artifact instances are usually called objects, and Artifact categories are Object Types. Examples of model languages are UML, SysML, Domain Specific Languages (DSL)…Although some of these formalisms used to be partly diagram based, there is currently an effort to provide them with a complete syntactic definition.
3. Definition of Domain Specific Language (DSL) is a particular kind of model.
   At this step, we do use two main notions:
   DSL types model artifacts categories
   DSL terms model artifact instance and get a DSL type (i.e., DSL term 'Steam Generator' of *DSL type 'System'*). In our process, we will consider a particular DSL that is simple enough to ease translation into a theorem prover and powerful enough to capture systems complexity.
   If we first define a model, we need to translate it into our DSL. The translation will associate to an object a DSL term and to an object type a DSL type.
4. The next step is the translation of the DSL into COQ.
   In COQ also, we can distinguish between terms and types.
   In the paper, we show it's possible to translate the DSL types into a single Inductive Type that we will name COQ_DSL. COQ_DSL is, in fact, a mutually inductive type, and each mutually defined inductive type in COQ_DSL corresponds to a single DSL type. So it's possible to make a one-to-one correspondence between DSL types and some COQ types.
   Once this is done, DSL terms can also be translated into COQ terms (see Fig.2).
   At this step, the System Model is defined as a set of artifacts with DSL types assigned and translated to COQ terms with appropriate inductive type constructors.

Fig. 2. Commutativity diagram between DSL and COQ terms

5. Finally, we can specify and prove the properties of the system and its artifact. The main task at this step is to translate system and model properties and rules into COQ theorems and prove them.

3 Model example

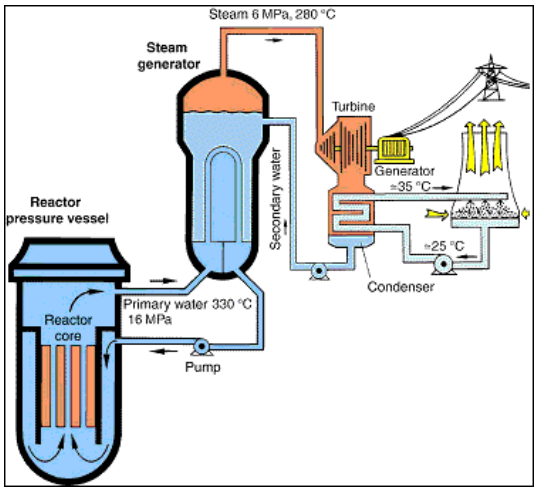

Fig. 3. Nuclear plant principle

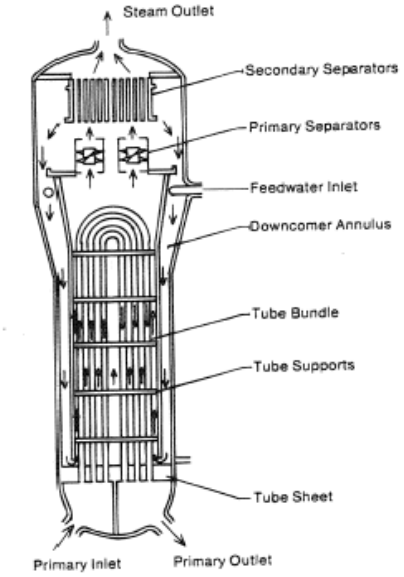

Fig. 4. Steam generator preliminary design

As example of a system model to be described, we took the general Nuclear Steam Generator (fig. 3,4) description represented in the AFCEN* guide [12]. Steam generators are heat exchangers used to make steam from water with the heat produced in a nuclear reactor core and are used in pressurized water reactors between the primary and secondary coolant loops. This is a good example of a system with a lot of safety requirements, a lot of physical relations between components, and some functional data flow between them. We will introduce DSL (Domain Specific Language) to describe the Generator model and then translate it into a formal description.

* AFCEN – French Association for Design, Construction and In-Service Inspection Rules for Nuclear Steam Supply System Components

In describing this example, we need to select the formal syntax for denoting system items and their hierarchical relationships and the formal syntax for denoting relations between system items. We can then introduce some inductive breakdown structure denoting the system as a whole. In our particular example, we introduce several Object Types like 'System', 'Components', 'Function', 'Requirement', and several categories for flows/relationships between these Objects Types: 'Data flow', 'Physical flow', and 'Refine'. Some system logic terms are also defined: 'System' can contain objects of 'Component' only, 'Component' can contain 'Component' or 'Function', and so on... In fig.5, we provide a simplified model implemented with the given Object Types above. 'Steam Generator Boiler' of type 'System' is our system of interest; this object contains children object of type 'Component' exchanging flows of type 'Physical flow'. The example presented is simplified; it has only several objects on the diagram. Our real example is based on a Use Case in the AFCEN guide [12] for risk analysis of nuclear pressurized equipment and consists of 66 Components, 153 Functions, 160 Physical flows, and 495 Requirements.

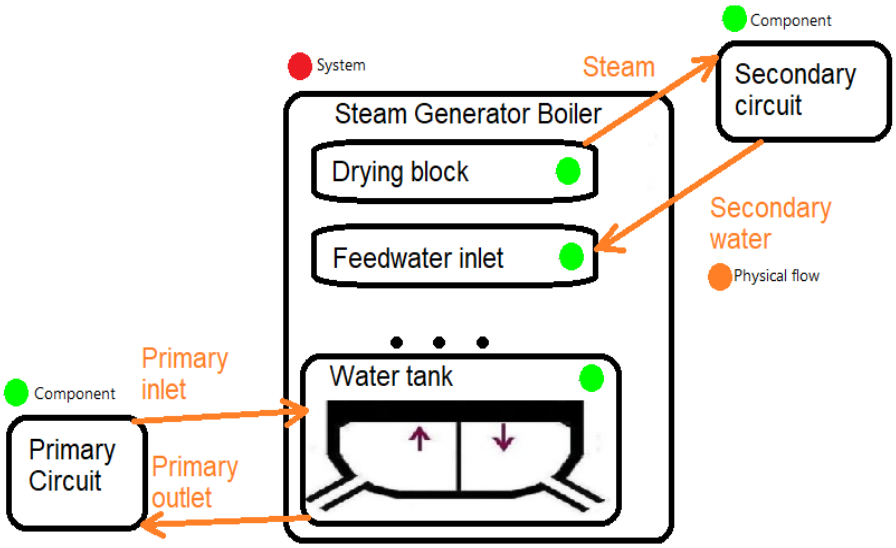

Fig. 5. Simplified model of Nuclear Steam Generator

4 Domain-Specific Language

Now we introduce the DSL supporting the syntactic description of our system through its hierarchy of objects as shown in Fig.5. Each object in the system model has a type, and hierarchical relations between types are strictly defined (some types can be recursive, e.g., a system may be composed of components and components may be composed of components). We propose interpreting these relations between DSL types as a Regular language (generated by type-3 grammar in Chomsky hierarchy [14]). I.e., if we have such a hierarchy of DSL terms: 'system 1' → 'function A' →'requirement X', which barely denotes that requirement X is allocated to function A that is allocated to the system, then an appropriate list of types for the

'requirement X' position in the hierarchy will be [system] — [function] — [requirement] and it is an accepted word for the language. Regular Grammar and Finite Automata(FA) being equivalent [1], we propose to describe our DSL language with the FA (∑, S, s0, δ, F) such that:

- ∑ – list of input alphabet = list of DSL types,
- S is a set of states (one state per DSL type),
- s0 – initial state (represents the root of objects DSL types hierarchy),
- δ is the state-transition function (displayed graphically in fig.6),
- F is the set of final states, equal to S to represent the simple DSL (hierarchical description in our example can be stopped at any level)

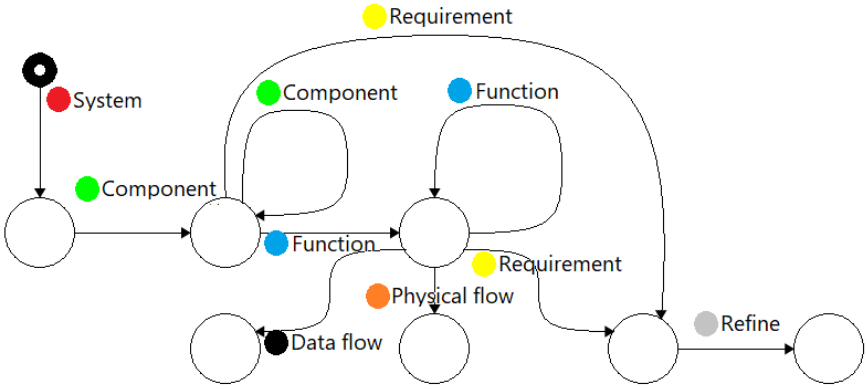

Fig. 6. Finite automata presentation of DSL types

Our sample DSL here is not universal. There can be other DSLs based on different types of grammar, some specific restrictions can be applied (like the rule 'each Component should contain a function), and so on. But most systems can be defined with such a simple breakdown. Note that DSL represented in Fig.6 is a type relation diagram, so having an arrow means that in a concrete example, there can be several children object instances for the given type (possibly no children).

At this step, our system of interest could be modeled either directly in a tool supporting the DSL in Fig 6 faithfully or by translating from an existing MBSE tool into our DSL. Our objective is to start from a DSL that is simple enough to enable a faithful translation into a theorem prover but smart enough to capture the complexity of a system model. The power of our approach relies on this trade-off between simplicity and faithfulness.

5 DSL model Translation

We show now that our DSL can be translated to an inductive type [8] which is a well-founded logical concept preserving the original semantics of DSL. It is possible to find a theorem prover supporting Inductive Types. We decided to use the well-known proof assistant COQ[6], based on the Calculus of Inductive Constructions (CIC). To reach our goal, we need to translate our DSL faithfully into COQ (according to Fig.2 in section 2), to define a list of formal rules that a system description in the DSL should comply with, and prove them with COQ tactics.

First, we define an inductive type “Direction” with possible values ‘input’ and ‘output’ to encode the direction of functional or requirement data flow messages. “direction” will be useful for any translation of DSL terms denoting messages or interactions between system items.

```
Inductive direction : Set := input | output.
```

Next, DSL types are translated into COQ types in a one-to-one manner, and these COQ types will be defined as a mutually inductive type that we call COQ_DSL. As an inductive type, each COQ type is built with a set of constructors.

Consider the DSL type ‘system’; the first constructor CSystem in Coq allows creating new COQ terms of type ‘system’. Then CComponent_System allows aggregating a set COQ_DSL ’components’ to a COQ_DSL ‘system’. CFunction_System allows aggregating ‘functions’ in a ‘system’ and CSystem_System allows adding a set of terms of type ‘system’ in a ‘system’ (fig. 7). The constructors mimic the way we can build systems with the DSL and translate them into COQ.

With this translation specification in mind, we developed an automated generation of the type system in Coq based on a DSL with a one-to-one translation of DSL types into COQ types. Below you will find a pseudo-code example for translating a DSL into a COQ script file. Assume that all DSL types exist as a list in the typeCollection, and we have functions to get a list of children DSL types and to understand if the relation describes some interface (flow).

This way, the definition of DSL described in the first part of the paper can be translated into an inductive types in COQ, as presented in Fig. 7. On the image, you can see the top DSL type ‘System’, with all related DSL types reminded regarding each type in COQ (see fig.6 for DSL type relations). The ‘with’ keyword in the script remind all types are mutually inductive. We use natural numbers to encode unique identifiers assigned to each type, this is done to support the ability to distinguish objects instances. Object attributes describing some properties can be assigned similarly; a small example of properties will be shown in the next section.

```
Definition Identifier : Type :=  N.
Inductive System : Type :=
| CSystem : Identifier -> System
| CComponent_System : list Component -> System -> System
with Component : Type :=
| CComponent :  Identifier -> Component
| CComponent_Component : list Component -> Component -> Component
| CRequirement_Component: list Requirement -> Component -> Component
| CFunction_Component : list Function -> Component -> Component
with Function : Type :=
| CFunction :  Identifier -> Function
| CFunction_Function : list Function -> Function -> Function
| CRequirement_Function : list Requirement -> Function -> Function
| CData_flow_Function : list Data_flow -> direction -> Function -> Function
| CPhysicla_flow_Function : list Physical_flow -> direction -> Function -> Function
with Requirement : Type :=
| CRequirement : Identifier -> Requirement
| CRefine_Requirement : list Refine -> direction -> Requirement -> Requirement
with Data_flow : Type :=
| CData_flow :  Identifier -> Data_flow
with Physical_flow : Type :=
| CPhysical_flow : Identifier -> Physical flow
with Refine : Type :=
| CRefine : Identifier -> Refine.
```

System
Component
Function
Requirement
Data flow
Physicla flow
Refine

Fig. 7. Translation DSL into COQ inductive types

6 Translation of system model to COQ

Having defined the translation of DSL types into COQ types, we now address the translation of DSL terms into COQ terms. We do use a recursive bottom-up algorithm starting from leaf DSL items and then translating upper DSL items with the set of children COQ terms already translated. After translation, we do compile the resulting files into a COQ module. This step is already a proof of correctness of the export process because COQ will warn if any inconsistency is found between translated DSL terms and types.

In the example given (fig.8), you can see the part of the hierarchy of objects and the flow input (the real example of this system contains more than 1000 lines in a COQ file):

- Steam_Generator_Boiler is an object of type System, is a parent of several Components: Drying_block, Feedwater_inlet, Water_tank (and some others in real code)
- Drying_block in an object of type component, it has some children of type Function and some children of type Component
- Feedwater_inlet is also an object of type Component, consuming 'Secondary_water' flow of type 'Physical flow.'

```
...
Definition Feedwater_inlet_27 := CFunction_Component [ ... ]
  (CPhysical_flow_Component [ Secondary_water; ... ] input
  (CComponent 2194728288258))).
...
Definition Drying_block_11 := CFunction_Component [... ]
  (CComponent_Component [... ]
  (CComponent 1103806595096)).
...
Definition Steam_Generator_Boiler_1 := CComponent_System
  [ Drying_block_11;
    Feedwater_inlet_27;
    Water_tank_32;
    ... ]
  (CSystem 1013612281857).
```

Fig. 8. shortened representation of data model in the COQ

The following pseudo python code is intended to walk through all objects of DSL types and generate appropriate commands into the COQ script resulting in the definition of the system with Coq inductive type:

```
Walk recursively through hierarchical object structure:
 When all children are parsed:
   Write 'Definition ' + object name + ':='
 For each child type:
   Write Child Type list Constructor +
```

```
    List of children of the given type
Write Object Type constructor + object Identifier
Write '.'
```

To check that translation of the system model is robust, we did define a backward-translation automatic script, converting COQ types into DSL and creating a DSL terms model according to COQ data. To make a fair transition, this script calls COQ to load modules containing data and reads defined type contents with COQ 'Print' command. It allows us to compare initial and resulting data in the system definition tool we use to show that all translations are bijective.

7 Verification

Once our system description is translated into COQ, we benefit from the advantages of a theorem prover. We can formalize properties that should be met by the system, hypotheses and demonstrate these properties.

For example, let's consider a property about data flows between Systems, Components, and Functions. An expected property of flows in the data model is that each produced flow should be consumed somewhere and conversely in the hierarchy of instances. Let's determine how to translate this property into COQ and make the proof that the model satisfies this property. We need to define the predicate as "any consumed data is produced" ("any produced data is consumed" is defined in the same way). Before specifying the goal, we need to create some recursive functions, walk through the hierarchy and look for flow objects. Here is the automatically created set of functions required for performing the verification (we display only a part of the proof).

"msgfunction" holds for a data d, a function g, and a production 'direction' (introduced in section 5) p if g contains (possibly in its subparts) the data flow d with direction p. Other definitions play a similar role for described COQ types. They study the given COQ type according to its definition and call appropriate functions recursively (to answer if the given flow with its direction is found in the term hierarchy or not):

```
Fixpoint  msgfunction (d:Physical_flow)(p:direction)(g:Function): Prop :=
match g with
|CFunctionMM n => False
|CPhysical_flow_Function listData prod obj => (In d listData)/\(prod = p) \/ (msgfunction d p obj)
...
end.
Fixpoint msgComponent (d:Physical_flow)(p:direction)(g:Component){struct g}: Prop :=
match g with
| CComponent n => False
| CFunction_Component lf s => ( fold_left or (map (msgFunction d p) lf ) False ) \/ (msgComponent p s)
...
end.
Fixpoint msgsystem (d:Physical_flow)(p:direction)(g:System){struct g}: Prop :=
match g with
| CSystemMM n => False
| CComponent_SystemMM lf s => ( fold_left or (map (msgComponent d p) lf ) False ) \/ (msgsystem d p s)
end.
```

Finally, we can describe the goal, a property that can be proved by Coq tactics. Normally

the definition and Goal below should be understandable intuitively from the previous definitions.

```
Definition EveryConsumedDataIsProduced(s: System) :=
forall (d:Physical_flow), (msgsystem d input s)-> (msgsystem d output s).

Goal (EveryConsumedDataIsProduced Steam_Generator_Boiler_1).
intro. simpl. tauto.
Qed.

Definition EveryProducedDataIsConsumed(s: System) :=
forall (d:Physical_flow), (msgsystem d output s)-> (msgsystem d input s).
```

With our experiments, we found that if the model is correct, it can be proved by Coq with simple tactics (automatically, as the set of tactics required is trivial), in case of missed producer/ consumer COQ will simply warn about the failed theorem proof. In that case, we need to perform some investigation to pinpoint the exact problem in the model. To find the problem, we wrote a script for an 'interactive' search for the problem in COQ terms, calling some COQ commands to perform checks and generating the next request with defined patterns.

8 FMEA

One interesting application of proofs of correctness and completeness is the safety (hazard) analyses. There are several hazard analysis methods available for investigating possible failures and describing ways to reduce risks. The most known ones are Failure Mode and Effect Analysis [3] and System Theoretic Process Analysis (STPA). The FMEA safety analysis method uses a physical component diagram (splitting the investigated system into parts, starting from low-level components and proceeding up to the failure effect of the overall system). At the same time, SPTA is based on a functional control diagram, analyzing the dynamic behavior of the system [2]. FMEA is an inductive procedure summarizing upward effects on the system from subsystems. So, the FMEA analysis is very well-structured, allowing to use of some taxonomy to define the order of analysis actions and relations between them. A good example of detailed taxonomy and conditions to be verified can be found in [7].

The DSL proposed in the first part of the paper is designed to describe the hierarchical decomposition of a system into basic elements, which is the source for FMEA analysis. In the image below, you can see the standard FMEA table according to (AFCEN guide [12]) with some assigned DSL types for objects (with some properties added), representing data for columns. The table (fig.9) is generated according to hierarchical object data stored in the system model.

Component Function Failure mode Failure effect Cause Requirement Risk reduction means

Component	Function	Failure mode	Failure effects			Essential safety requirements (order, decree)	Risk reduction means	
			Description	Pressure / Radioprotection risk	Phase		Proof means	Proof
U-tubes&Tube Sheet welding	Confine fluids in all Temp/Pressure situations	Plastic Instability	Loss of containment of Primary fluid in any situation	Pressure risk	Design	A1-2 A1-3.6	Use of a proven calculation method	Dimensioning reviews
U-tubes&Tube Sheet welding	Confine fluids in all Temp/Pressure situations	Fast fracture	Leak of radioactive sustances	Radioprotection risk	Procurement	A1-3.1 A1-3.2 A1-3.5 A1-4.3	Drafting a procurement specification for nuclear pressure	Supply specification
U-tubes	Confine fluids in all Temp/Pressure situations	Fast fracture	Leak of radioactive sustances	Radioprotection risk	Procurement	A1-3.1 A1-3.2 A1-3.5 A1-4.3	Drafting a procurement specification for nuclear pressure	Supply specification

Fig. 9. FMEA tale with DSL language types assigned

Having the data structure required and FMEA logic, we can define some completeness formal rules to prove in COQ. As an example of property defined by law, we consider the classification of equipment parts (depending on their role with regards to pressure resistance) defined by Autorité de sûreté nucléaire (Nuclear Safety Authority, ASN), represented in fig.10 (shortened comparing to [9]):

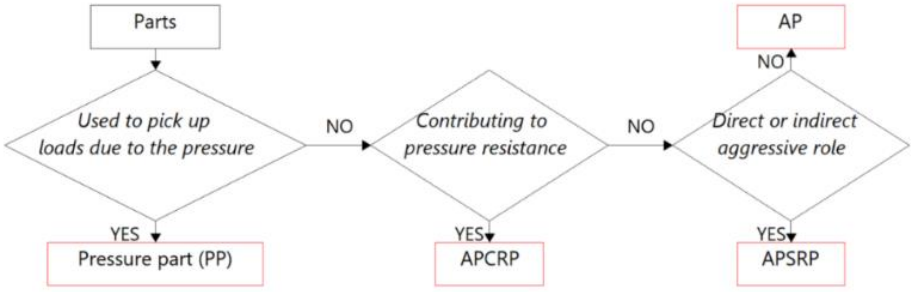

Fig. 10. Component classification logic

To represent those properties in our model, we need first to define appropriate attributes for each Component object (with a defined list of possible values for each attribute):

- ‘is used to pick up loads due to the pressure’ – yes/no,
- ‘is contributing to pressure resistance’ – yes/no,
- ‘is direct or indirect aggressive role’ – yes/no,
- ‘classification’ – PP/AP/APCRP/APSRP.

Then those attribute values can be translated to the coq model as several sets and assigned

to object instances:

```
Inductive PressureLoads : Set := P_yes | P_no.
Inductive PressureResistance : Set := R_yes | R_no.
Inductive DirectAggressiveRole : Set := A_yes | A_no.
Inductive ObjType : Set := APCRP | PP | APSRP | AP.
Inductive Component : Type :=
| CComponent : nat -> PressureLoads -> PressureResistance
    -> DirectAggressiveRole -> ObjType -> Component
| CFunction_Component : list Function -> Component -> Component.
Definition Component3 := CComponent 3 P_no R_no A_yes AP.
```

From a practical point of view, identifying all pressure parts requires identifying parts that have an interface with primary or secondary circuits water and steam. So starting from a specification of all systems and parts interfaces, it is possible to formally identify all PP parts from the Coq Model. Explaining this in detail goes beyond the objective of this paper, but this is a good example of a part of the safety case that could be supported automatically and faithfully by a proof system. Of course, the identification of interfaces is based on systems engineer analysis and may be the weak spot of the proof.

9 Conclusion

The paper shows how system models can be translated into COQ definitions. Making proofs using only system descriptions allows us to work on a high abstract level (without detailed system behavior descriptions). This is especially major for investigating the correctness and completeness of the system with bottom-up safety analysis like FMEA.

Using a general system definition language (DSL) including recursive types definition, we did turn the DSL into a COQ inductive types; having this, we can translate the syntax of the system model (DSL terms) into definitions of COQ terms. Starting from system properties to be proven, we can translate them formally into COQ theorems and prove them either automatically or not. COQ is also capable of finding counterexamples in many cases if a property is not true (a small description is given at the end of section 7).

With this approach, systems and safety engineers get the possibility to back their systems and safety analysis with formal proofs of correctness and completeness that have to be stated in parallel.

Now we plan to investigate further topics to push the limits on this approach - investigate the completeness and correctness properties that systems and safety analysis should comply with. We started with simple properties in this paper, but generally, the Safety case comprises high-level goals that are decomposed into simpler elementary goals and action plans. We will investigate the formal rules behind such a decomposition. In parallel, we will look for a formal analysis regarding the completeness and correctness of a system design.

Then we need to check the resilience of a proof system like Coq when investigating models made of 10th or 100th thousands of objects, as this is the size of the complex system models we discuss.

After that, we would like to investigate the user's assistance when performing a system or safety analysis. For instance, in case we already know the functional and system architecture of the system, we can check for completeness and correctness properties and raise anomalies to the user. In case a safety goal is being decomposed, we can identify automatically if all subgoals are covering all the possibilities or not.

Finally, we will investigate how to connect MBSE tools to our system and safety formal DSL description, allowing the formal models to be filled by an existing modeler. Of course, we will have to prove that the translation is faithful, which could be made by bi-directional translation between the MBSE and the formal DSL and proof of evidence that there is a bijection.

References

[1] Grune, D., Jacobs, C.J.H. (2008). Regular Grammars and Finite-State Automata. In: Parsing Techniques. Monographs in Computer Science. Springer, New York

[2] Sulaman, S., Beer, A., Felderer, M. et al. Comparison of the FMEA and STPA safety analysis methods–a case study. Software Qual J 27, 349–387 (2019).

[3] N. Storey, "Safety-critical computer systems", Addison-Wesley,1996.

[4] Gleirscher M, Foster S, Woodcock J (2019) New opportunities for integrated formal methods. ACM Comput Surv 52 (6):117:1–117:36.

[5] I. Lopatkin, Y. Prokhorova, E. Troubitsyna, A. Iliasov, and A. Romanovsky. Patterns for Representing FMEA in Formal Specification of Control Systems. TUCS Technical Report 1003, 2011.

[6] COQ, https://coq.inria.fr/documentation, last accessed 2022/03/29.

[7] E. Piljugin, S. Authén, J-E. Holmberg. Proposal for the Taxonomy of Failure Modes of Digital System Hardware for PSA. 11th International Probabilistic Safety Assessment and Management Conference & The Annual European Safety and Reliability Conference At: Helsinki, 2012

[8] Pfenning, F., Paulin-Mohring, C. (1990). Inductively defined types in the Calculus of Constructions. In: Main, M., Melton, A., Mislove, M., Schmidt, D. (eds) Mathematical Foundations of Programming Semantics. MFPS 1989.

[9] Conformity assessment of nuclear pressure equipment. Autorite de surite nucleare. Guide N8. 2012.

[10] S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. In FM 2006: Int. Symp. on Formal Methods, volume 4085 of LNCS, pages 460–475. Springer, 2006.

[11] National Institutes of Standard Technologies, Planning Report 02-3 The Economic Impacts of Inadequate Infrastructure for Software, 2002

[12] Guide ADR (Analyse de risques) pour ESPN N1, AFCEN, 2018

[13] INCOSE Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities, 4th Edition, INCOSE, 2015

[14] Chomsky, Noam. "Three models for the description of language". IRE Transactions on Information Theory, 1956