|  |  |
| --- | --- |
| D2.1 | CRML methodology |
| Access[[1]](#footnote-1): | **PU** |
| Type[[2]](#footnote-2): | **Report** |
| Version: | **1.0** |
| Due Dates[[3]](#footnote-3): | **M12** |
| Environment for model-based rigorous adaptive co-design and operation of CPS | |
| **Executive summary[[4]](#footnote-4):** | |
| This document discusses the aspects of the CRML methodology related to assume/guarantee contracts and the articulation between SysML and CRML.  The outcome of this discussion will be used in the second version of the CRML specification. | |

**Deliverable Contributors:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Name | Organisation | Primary role in project | Main Author(s)[[5]](#footnote-5) |
| Deliverable Leader[[6]](#footnote-6) | Daniel Bouskela | EDF | WP2 Leader |  |
| Contributing Author(s)[[7]](#footnote-7) | Zoltán Micskei | BME | T2.1 member | X |
| Vince Molnár | BME | T2.1 member | X |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
| Internal Reviewer(s)[[8]](#footnote-8) | Lena Buffoni | LIU | Project Leader |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**Document History:**

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Reason for Change | Status[[9]](#footnote-9) |
| 1.0 | 15/01/2021 | First Issue | In Review |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Contents**

[Abbreviations 4](#_Toc61627093)

[1 Introduction 5](#_Toc61627094)

[2 SysMLv2 and CRML [2p] 5](#_Toc61627095)

[2.1.1 Modelling requirements and systems in SysMLv2 5](#_Toc61627096)

[2.1.2 Comparing SysMLv2 and CRML 7](#_Toc61627097)

[2.2 Use cases and analysis of A/G contracts 9](#_Toc61627098)

[2.2.1 The A/G Contract profile for SysML 10](#_Toc61627099)

[2.2.2 Considerations about A/G Contracts 10](#_Toc61627101)

[References 15](#_Toc61627102)

# Abbreviations

List of abbreviations/acronyms used in document:

**Abbreviation Definition**

A/G Contract Assume/Guarantee Contract

CRML Common Requirement Modeling Language

OMG Object Management Group

# Introduction

The specification of CRML consists in the description of the syntax of the language using a formal approach such as extended BNF (Extended Backus-Naur Form), and the formal description of the semantics (using mathematical notation). The syntax states the rules for forming valid CRML expressions. The semantics states in a formal way using 4-valued Boolean algebra when requirements are satisfied. The language will carry sufficient information so that it will be possible to perform the automatic CRML models processing planned in tasks T2.2, T2.3 and T2.4.

The methodology shows how to use the language and associated tools (graphic interface, A/G contracts) to build and ensure the traceability of the specifications, perform verifications and optimizations throughout the entire system lifecycle.

The specification and the methodology will be tailored to satisfy the end-users needs as expressed in T2.5.

# SysMLv2 and CRML [2p]

CRML is intended to be used in connection with other modeling languages and tools. This connection can be manifested in several forms: referencing various models or model elements representing other parts of the system, transforming parts of the CRML model to other formalism, etc. See D2.2 for a more detailed description of possible connections and tooling support.

One of the prime candidates for such connections is **SysML**, the OMG Systems Modeling Language [OMG-2019]. SysML is a modeling language for systems engineering with widespread tool support and adoption in several domains (e.g., automotive, aerospace, railway). SysML defines languages for modeling requirements, structure (blocks, parts and ports) and behavior (state machines, activities and interactions). Describing requirements and various constraints is highly relevant for CRML for interoperability reasons (i.e., defining bindings later for SysML), therefore this section overviews and compares the modeling elements and expressive power of both languages.

The current version of SysML is v1.6. However, OMG published a request for proposal for the next version of SysML (named SysMLv2). A draft version has already been prepared and published online [SST-2020]. As the intended submission of the final version is in the timeframe of the EMBrACE project, we focus on this next version to make the contributions of the EMBrACE project future-proof.

First, this section presents an overview of SysMLv2 concentrating on the parts relevant to CRML. Next, we compare the modeling elements of SysMLv2 and CRML.

### Modelling requirements and systems in SysMLv2

SysMLv2 is a general-purpose modeling language that is based on the new Kernel Modeling Language (KerML). SysML also includes a textual notation besides the diagramming elements of SysMLv1. The following text is based on the 2020-11 release of SysMLv2 [SST-2020].

SysMLv2 groups the modeling language in several packages (see Figure 1): ranging from basic model elements (e.g., *Classifier*, *Attribute*) to more systems-specific constructs (e.g., *Part* or *Port*). A main design pattern used consistently throughout the language is *Definition* and *Usage* that supports reuse. A definition element classifies certain kinds of elements, while a usage element “is a usage of a definition element in a certain context” [SST-2020]. For example, a part definition can represent a certain type of flow sensors, and that sensor can be used in various roles in the modelled system (e.g., measuring the in or out flow of a tank).

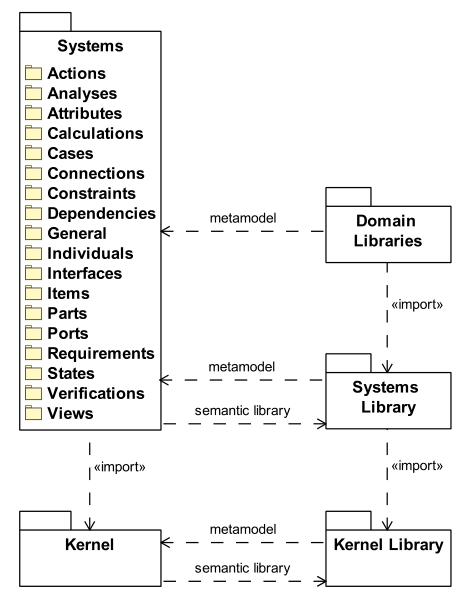


Figure 1. SysMLv2 Language architecture [SST-2020]

The main modeling elements relevant for CRML are the followings.

* *Constraint*: a constraint definition is a parametrized Boolean predicate that defines an expression to constrain some feature of the model. A constraint definition can be used, where its parameters are bound to some parts of the model. General constraints can be satisfied or violated during the lifetime of the system. However, asserted constraints represent invariants, which cannot be violated by a valid model.
* *Requirement*: a requirement definition is a stakeholder-imposed constraint that the model shall satisfy to be treated as a valid solution. Requirements can include textual statements and formally specified constraints, too. Requirements can be hierarchical. Requirements have unique identifiers and can define a subject (which can be any model element: parts, interfaces, and even actions). A requirement can also contain assumptions (special constraints).
* *Case*: analysis and verification cases can be defined that represent the context where requirements should be evaluated. Analysis cases can specify complex calculations that are solved by external solvers. Verification cases define the steps needed to verify whether a system satisfies its requirements.

The following example illustrates these general concepts (taken from the training material of [SST-2020]). The Vehicle part definition represents a general vehicle with dry and fuel mass. Requirement 1 is a general requirement definition that actual mass of a vehicle (its dry mass plus the mass of its current fuel) shall be less than or equal to a predefined required mass. Requirement 1.1 is a requirement usage defining that if the fuel tank is empty (assumption) then the required mass is 1500 kg.

**part** **def** Vehicle {

**attribute** dryMass: MassValue;

**attribute** fuelMass: MassValue;

…

}

**requirement def id** '1' VehicleMassLimitationReq {

**doc** /\* The total mass shall be less than or equal to the required mass.\*/

**subject** vehicle : Vehicle;

**attribute** massActual = vehicle::dryMass + vehicle::fuelMass;

**attribute** massReqd : MassValue;

**require** **constraint** { massActual <= massReqd }

}

**requirement** **id** '1.1' emptyVehicleMassLimit : VehicleMassLimitationReq {

**subject** vehicle : Vehicle;

**attribute** **redefines** massReqd = 1500@[kg];

**assume constraint** {

**doc** /\* Fuel tank is empty. \*/

vehicle::fuelMass == 0@[kg]

}

}

### Comparing SysMLv2 and CRML

The most important difference between CRML and SysMLv2 is the main purpose of the modeling languages. While SysML aims primarily to precisely specify and integrate system components, a strong motivation for CRML is to leave the specification as loose as possible, that is, to define a *design envelope*. This difference in philosophy comes from the fact that CRML is not created to precisely design implementations, but rather to capture the assumptions and requirements related to the system without forcing the engineers to make premature design decisions. Furthermore, CRML is designed with a heavy focus on cyber-physical systems (CPS), where systems coexist and interact with their physical environment.

Therefore, CRML has stronger modeling capabilities when it comes to uncertainty, nondeterministic behavior and features language elements suitable to specify complex time-dependent and continuous behavior. Although modeling acausal relationships in SysML is also possible, it is not trivial to use the language for such purposes. The expression language and the *constraint*, *requirement* and *assumption* elements in SysMLv2 improve the language in this direction, but designing CRML to be interoperable with SysML would be the best way to combine the benefits of both languages.

A more detailed comparison of the languages can be seen in Table 1.

|  |  |
| --- | --- |
| Aspects of CRML | Remarks |
| Time intervals | SysMLv2 does not feature a precise semantics for time intervals. CRML has operators for combining time intervals in several ways. |
| Clocks | SysMLv2 has timed events, but does not have an explicit notion of clocks. CRML has continuous and discrete clocks that can be defined in the model. |
| Events | Both languages have a notion for events, but SysMLv2 has no features reasoning about event occurrences in a precise way. CRML offers several operators over events and allows the definition of time intervals with events, as well as filtering events with time intervals. |
| Requirement | SysMLv2 also has requirements, but they are not temporal logic expressions like in CRML. CRML also features operations over requirements, while SysMLv2 has composition and assumption only. CRML has a precise semantics for the evaluation of requirements based on a 4-valued Boolean satisfaction model. |
| Variables | Both languages have a notion for variables, and they both support the most important domains. |
| Operators | Both languages feature various operators over many domains. One of the most important improvements of SysMLv2 over SysMLv1 is the introduction of an expression language. |
| Requirement templates | Both languages feature parameterizable requirements, although CRML has a more powerful language with various operators over requirements. |
| Sets | SysMLv2 introduced individuals, which represents a specific entity of a certain type. With this and multiplicities combined, both languages are capable of representing sets, although CRML has dedicated operators to quantify and manipulate them. |
| Classes and objects | Both languages have a notion of classifiers and objects, although the philosophy differs here. SysMLv2 prefers *usages* instead of objects and has an extensive set of modeling elements that can be used to model structure. CRML does not aim for the precise design of system structure, its main goal is to capture the significant entities and variables necessary to specify the system. |
| Physical quantities | Both languages support the definition of physical quantities with units of measurement. |
| Behavior | CRML does not support the direct modeling of concrete system behavior. Instead, it supports the specification of such behavior with constraints and assumptions in a more declarative way. However, CRML supports the notion of bindings to get information from system behavioral models. SysMLv2 puts a heavy emphasis on the precise definition of system behavior with several diagram types. The connection here will be important in V&V activities. |

Table 1: Comparison of CRML and SysMLv2

As we can see from the table, the most likely and natural connection between CRML and SysMLv2 will be a *specification-design* relationship. We plan to define bindings between the more complex structural elements of SysMLv2 and the simplistic notion of classes and variables of CRML to enable the interpretation of CRML requirements on concrete SysMLv2 design models. Evaluation of a requirement can then be performed through the simulation of SysMLv2’s complex behavior models or by formal verification. Technically, CRML will be an additional constraint language applicable in SysMLv2, focusing on specifying design envelopes and CPS-related aspects.

## Use cases and analysis of A/G contracts

In addition to designing CRML to be compatible with SysML, we also extend the notion of requirements in SysML. Currently, SysML does not differentiate *assumptions* and *guarantees*, it only knows *requirements*. Even though a requirement may have an assumption, this feature is not related to the *A/G Contract* concept that we aim to introduce.

A/G Contracts not only specify requirements towards a system or its components, but they also facilitate the even more critical communication, or *bargaining,* so to say, that occurs in the early stages of system specification between different stakeholders responsible for subsystems. In the final design, there is no need to split a requirement into assumptions and guarantees, because the requirement itself will represent both. During design, however, guarantees and assumptions typically do not match immediately. Obviously, if the assumption is stricter than the guarantee, they design will not work, so stakeholders have to discuss how to arrive to a requirement that is suitable for both parties. Having a stricter guarantee, on the other hand, is also worth consideration, because it may open up possibilities for optimization, or even mean the solution for a mismatch between some other parties by reallocating surplus resources to where they are more needed.

Therefore, we find it highly valuable to allow the representation of intermediate assumptions and guarantees during system design instead of rigidly defined requirements. This part of our work considers SysMLv1 for practical reasons, but the approach is compatible with SysMLv2, too. For the details about the implementation of this extension, see D2.2.

### The A/G Contract profile for SysML

Interfaz de usuario gráfica, Aplicación, PowerPoint

Descripción generada automáticamente

### Considerations about A/G Contracts

During the design of the profile extension, we have run into a design question about the multiplicity of assumptions and guarantees belonging to a contract. The question is related to the intended semantics of contracts, and our considerations will be presented in this section.

#### Multiple assumptions, multiple guarantees

The most generic solution is to allow the modeler use multiple assumptions and guarantees in a single contract. This gives a flexible language, and tools interpreting the contracts may impose their own limitations if necessary. However, thinking about semantics, this solution may lead to a lot of ambiguity. Here we describe three possibilities.

The most natural way to interpret such a contract is to say that the conjunction of every guarantee shall imply *each* assumption separately. This means that the contract groups all the assumptions we have to make about the environment of a component, and we collect every guarantee that is relevant to the design of this component. It is possible to split the contract into smaller contracts with only a subset of the assumptions and either all guarantees or only those that are necessary to satisfy the assumptions.

Another interpretation could be that the conjunction of guarantees should imply the *conjunction* of assumptions. With this, we would essentially define a composite assumption, which is the intersection of all its constituent assumptions. While this may be reasonable at times, it is basically the same as having a single assumption, given that we have support for combining requirements with at least the conjunction operator.

A third alternative is the generalization of the second: a logical expression over guarantees should imply a logical expression over assumptions. Again, this is similar to having a single assumption and a single guarantee, with support for the logical combination of requirements (which CRML has).

#### Single assumption, multiple guarantees

As we saw, the most natural interpretation of contracts allows the refactoring of a complex contract into a separate contract for every assumption. The advantage of this would be the explicit semantics, as the limitation of assumptions essentially discard every other interpretation (except that we take the conjunction of guarantees). A disadvantage is the increased size of the model, which may hinder readability.

#### Single assumption, singe guarantee

The advantage of this version is that it is trivial to define the semantics: the guarantee should imply the assumption. An easy-to-see drawback is that it limits the modeler, and does not allow for multiple guarantees satisfying an assumption. This would mean that an assumption of one party may not be satisfied by the cooperation of multiple other parties, but they would have to make separate agreements. To evaluate the significance of this limitation, we illustrate three examples.

##### Example: Weights

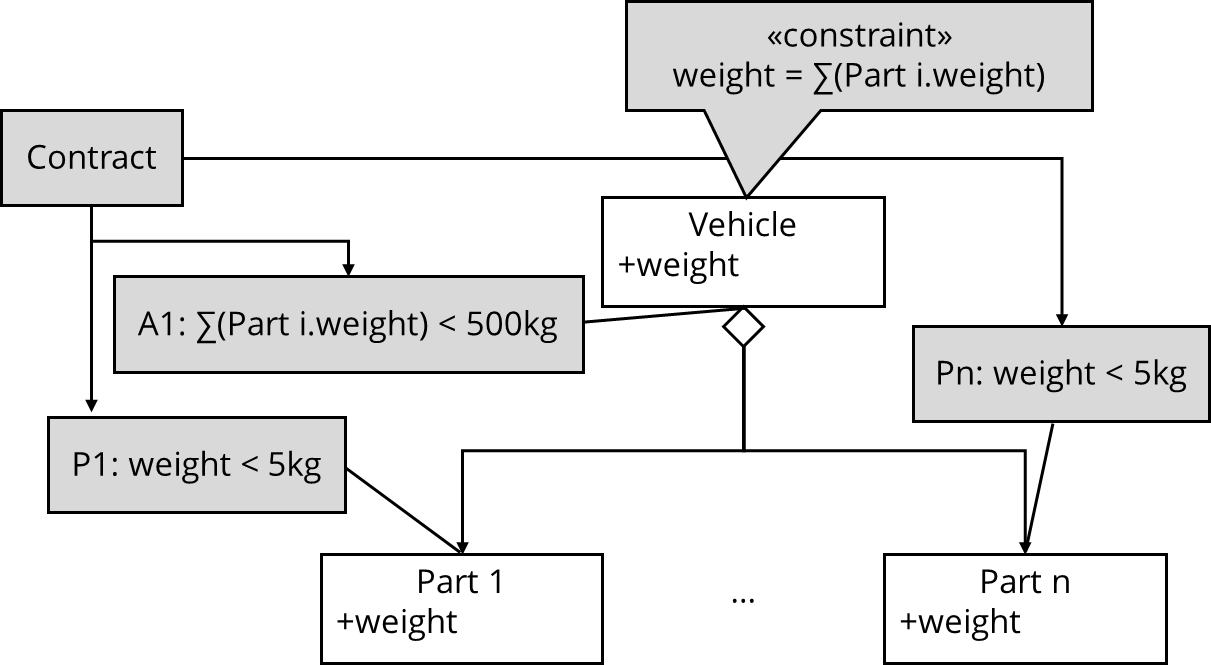


Figure 2. Single contract for weights

Figure 2 presents a schematic model describing the weights of a vehicle and its parts. If we allow multiple guarantees, the negotiation of individual part weights can be captured with a single contract: each part *promises* (hence the letter P for guarantees) the limit on its own weight, then the vehicle formulates an assumption on the sum of weights. A solution with the single-assumption-single-guarantee approach can be seen in Figure 3.

Comparing the approaches on this example shows that the main difference is not necessarily a problem of modeling, but more related to the *process*. If we assume that part vendors are willing and able to cooperate to satisfy the assumption (e.g., they are part of the same company), then the first solution is better. However, if vendors communicate only with the vehicle manufacturer, bargaining should be about separate contracts, and the coordination of this process is the job of the vehicle manufacturer. This latter assumption seems more realistic.

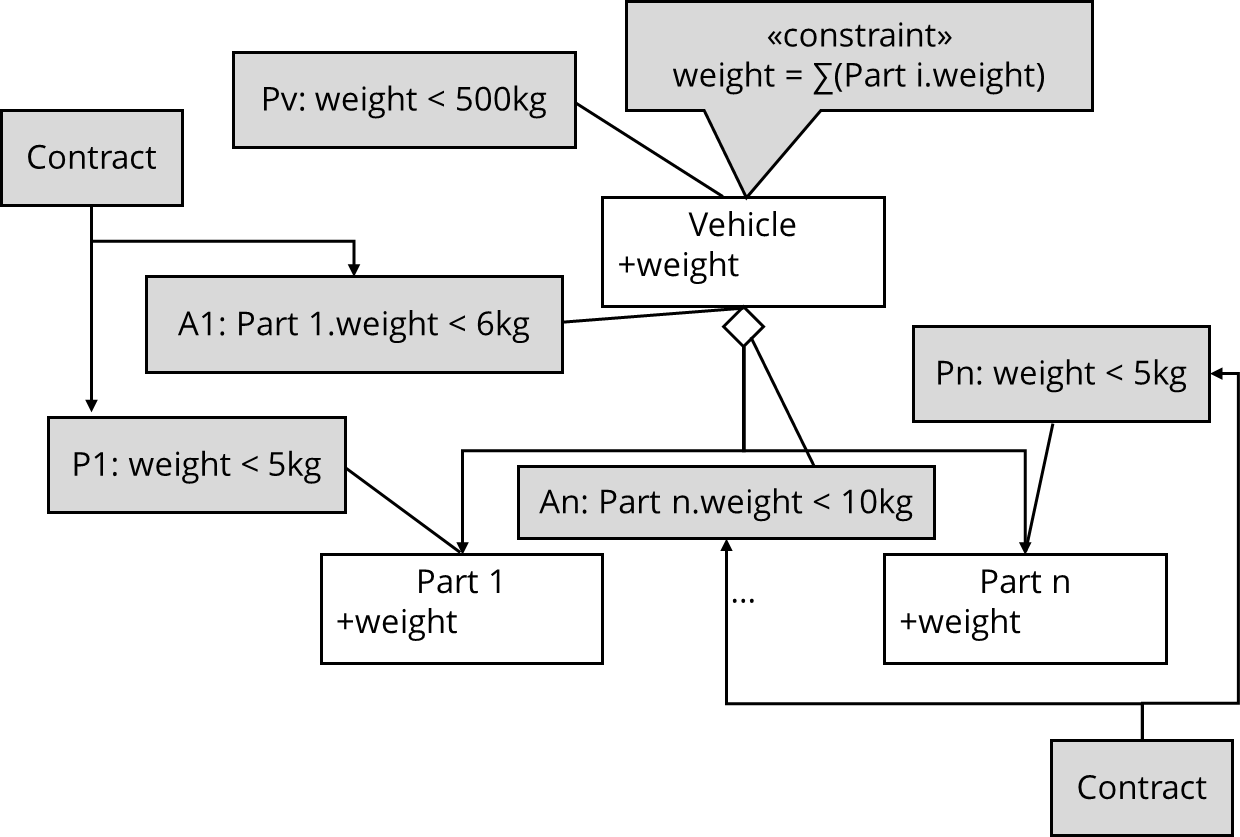


Figure 3. Multiple contracts for weights

##### Example: Communication bus

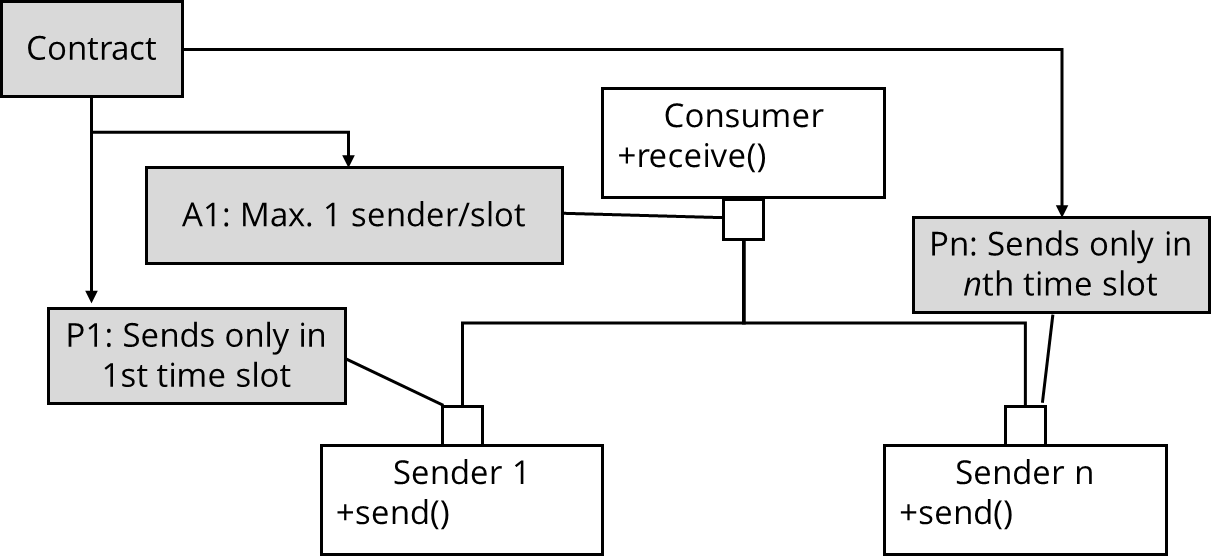


Figure 4. Single contract for communications

A very similar problem can be seen in Figure 4. Here, the single assumption is captured by a component consuming messages from multiple sources, and states that at most 1 component may send a message in a given time slot. The guarantees are about the time slot a component will use. Again, a solution with 1-1 contracts is in Figure 5.

The situation is similar here: if we assume that the components have a way to organize their schedule, then the consumer may have a single assumption affecting all the senders. This would be reasonable when the set of senders are not known in advance, but then there would be a protocol responsible for assigning slots to new senders and that protocol would need to guarantee our assumption. If we use multiple contracts, we put the responsibility of assigning the time slots to the consumer component. In static settings, this option is again more realistic.

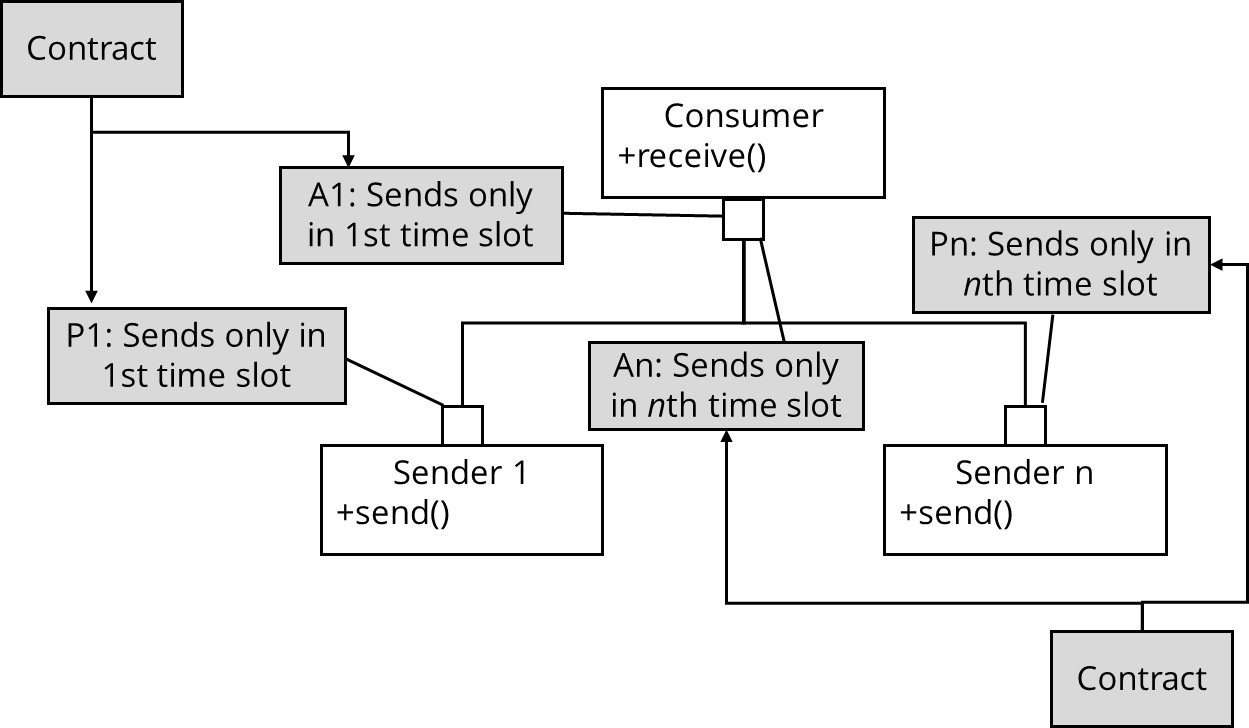


Figure 5. Multiple contracts for communication

##### Example: Monitoring

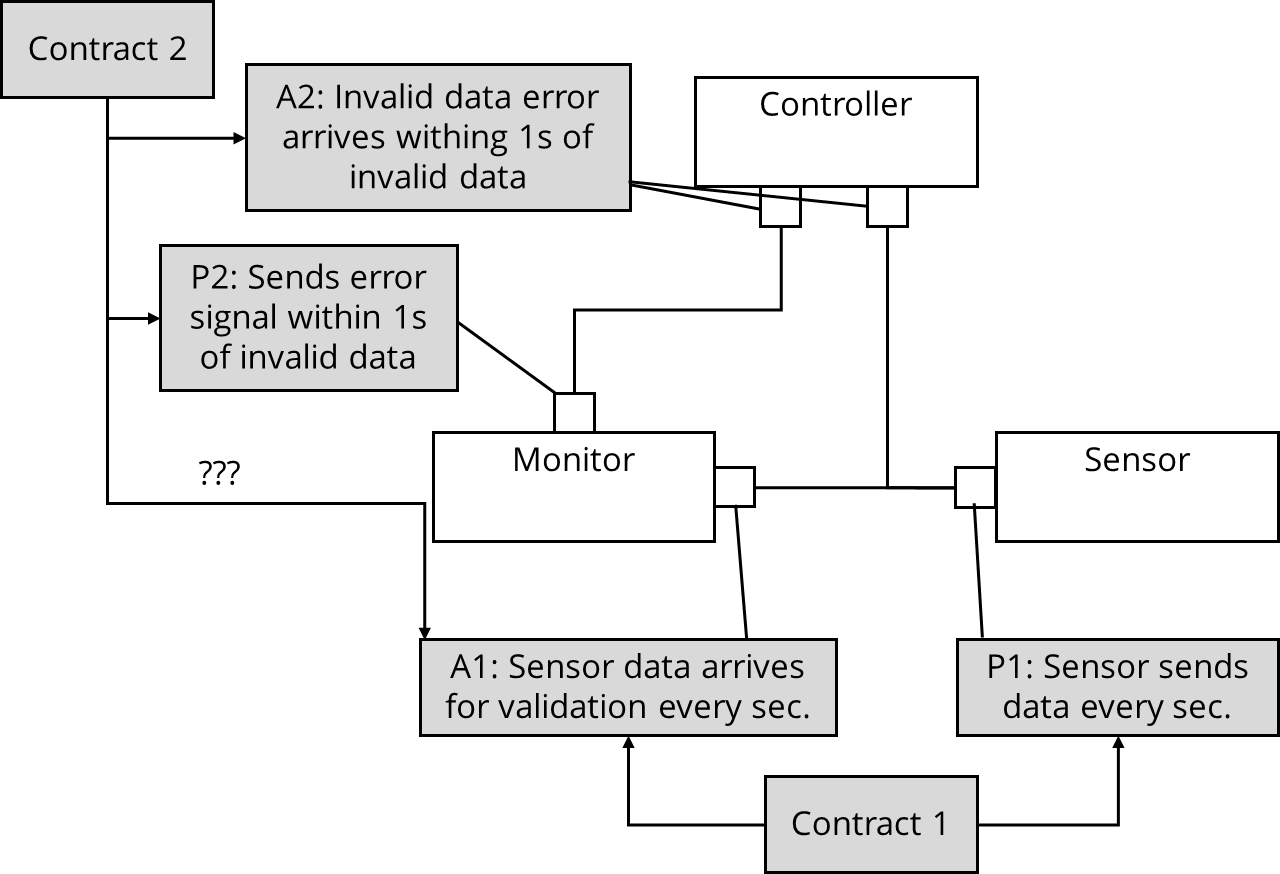


Figure 6. Contracts affecting multiple components

A more complex example is shown in Figure 6. Here, a *controller* is performing some action based on data from a *sensor* and while the action to take is computed based on the data, a *monitor* component also processes the data in parallel to see if it is valid. This cross-checking should happen before the actuation is performed, but the two calculations will happen simultaneously to increase performance. In this context, there is a contract between the sensor and the monitor about getting the date (and a similar contract could be between the controller and the sensor, too), and another, quite complex contract: the invalid signal from the monitor should arrive to the controller not later than 1s after the invalid data arrived from the sensor. At first, this seems to be a contract between the monitor and the controller, but in fact, there is some transitivity, because we need to assume that the contract between the monitor and the sensor also holds. This raises the question of whether we need *hierarchical contracts*, whichwill be examined later.

#### Remarks on the current proposal

After the examination of the above examples, we are currently proposing to introduce the single-assumption-multiple-guarantees variant of A/G contracts. Furthermore, the most recent discussions have raised the idea that contracts should not be defined between *components*, but between *stakeholders* [Azzouzi]. This shift in view would solve most of the problems in the examples, and would essentially mean that assumptions and guarantees could reason about all the components belonging to a stakeholder. It would also more accurately represent our goal to facilitate *bargaining* between stakeholders, as the contracts would be directly bound to them. Lastly, it would allow the modeling of contracts about extrafunctional or project-related aspects like money- or time-budget.

# References

Provide reference list here. Use reference numbers in brackets [nn] listed in order of appearance, or apply alphabetically ordered Author-date referencing.

[OMG-2019] OMG. 2019. OMG System Modeling Language (SysML). formal/19-11-01.

[SST-2020] SysML v2 Submission Team (SST). 2020. OMG System Modeling Language™ (SysML®) v2 Release. URL: <https://github.com/Systems-Modeling/SysML-v2-Release>

Azzouzi E., “Multi-faceted modeling of multi-energy systems: Stakeholders coordination,” PhD Dissertation, U. Paris-Saclay, Jan. 2021 (to be published after defense).

1. Access classification as per definitions in PCA; PU = Public, CO = Confidential. Access classification per deliverable stated in FPP. [↑](#footnote-ref-1)
2. Deliverable type according to FPP, note that all non-report deliverables must be accompanied by a deliverable report. [↑](#footnote-ref-2)
3. Due month(s) according to FPP. [↑](#footnote-ref-3)
4. It is mandatory to provide an executive summary for each deliverable. [↑](#footnote-ref-4)
5. Indicate Main Author(s) with an “X” in this column. [↑](#footnote-ref-5)
6. Deliverable leader according to FPP, role definition in PCA. [↑](#footnote-ref-6)
7. Person(s) from contributing partners for the deliverable, expected contributing partners stated in FPP. [↑](#footnote-ref-7)
8. Typically person(s) with appropriate expertise to assess deliverable structure and quality. [↑](#footnote-ref-8)
9. Status = “Draft”, “In Review”, “Released”. [↑](#footnote-ref-9)