

# IBM Technical Report 2006

TR-20060603

Accepted to appear in Journal of Object Technology (JOT) at [www.jot.fm](http://www.jot.fm)

## An Overview of the Systems Modeling Language for Products and Systems Development

Laurent Balmelli, Ph.D.

[balmelli@us.ibm.com](mailto:balmelli@us.ibm.com)

IBM Research Division

T.J. Watson Center and Tokyo Research Lab

10/2/06 – rev 14.

**keywords: SysML, UML, product development.**

### Abstract

*In this paper we present an overview of the capabilities of the Systems Modeling Language (SysML.) SysML is a standard from the Object Management Group. It is geared toward incrementally refinable description of conceptual design and product architecture. Elements in the design represent abstractions of artifacts in the various engineering disciplines involved in the development of the system. The design represents how these artifacts collaborate to provide the product functionalities. This paper explores all the diagrams available in SysML through the real-life example of an embedded system.*

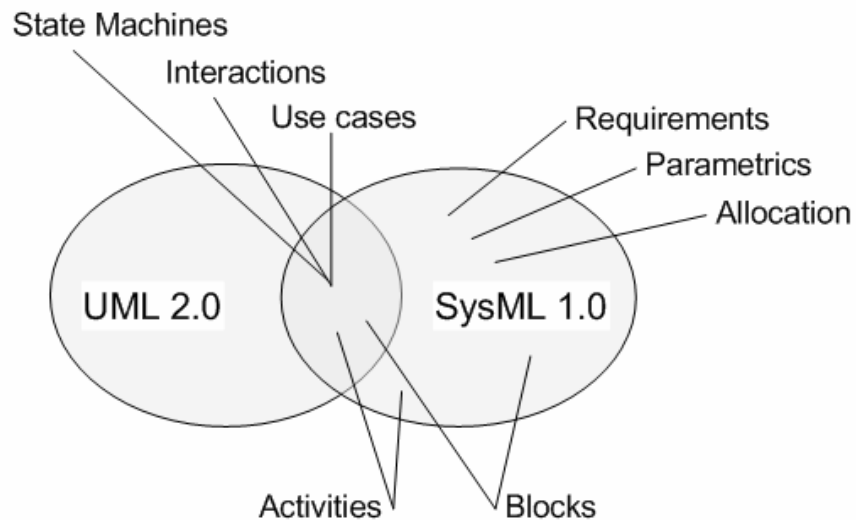
1. Introduction.....	2
2. Context, Requirement and Use Cases .....	3
3. Structure of the Rain Sensing Wiper system.....	8
4. Behavior.....	16
5. Allocation .....	22
6. Conclusion .....	23
7. References.....	24
Appendix A: the Rain Sensing Wiper Story .....	25
Appendix B: Additional Diagrams.....	26

## 1. Introduction

Today's competitive pressures and other market forces drive manufacturing companies to improve the efficiency with which they design and manufacture products and systems. Across the product lifecycle, one area where there has been a notorious lack of efficiency support is the conceptual stage, during which the functional architecture (and sometimes the physical architecture) is decided upon.

The conceptual stage follows the transformation of customer needs into product functions and use cases, and precedes the design of these functions across the engineering disciplines (for example, mechanical, electrical, software, etc.). A lack of support during product conceptualization makes it difficult to efficiently trace the realization of requirements in the product. The lack of a formal representation for concepts also results in an inadequate ability to make decisions at the level of systems in the product, such as during feasibility studies. Moreover, the lack of a clear vision of the product architecture hinders team understanding and communication, which in turn often increases the risk of integration issues. It is these and other challenges confronted during the conceptual phase of product and system development that SysML is designed to mitigate.

SysML is based on the actual standard for software engineering, the Unified Modeling Language (UML) developed within the Object Management Group (OMG) consortium. SysML was developed as a response to the request for proposal (RFP) issued by the OMG in March 2003. The development team includes representatives from more than ten companies. IBM has played a leadership role in the definition of the standard by authoring part of the specification.



**Figure 1 Comparison of SysML1.0 with UML2.0: The text in the figure summarizes the various diagrams available in SysML. Requirements, Parametrics and Allocations are new diagrams available only in SysML. Activity and Block diagrams are reused from UML2.0 and extended in SysML. Lastly, State Machines, Interactions and Use cases are reused from UML2.0 without modifications.**

SysML is a modeling language for representing systems and product architectures, as well as their behavior and functionalities. It builds on the experience gained in the software engineering discipline of building software architectures in

UML (think of the classic Class Diagram.) The architecture represents the elements realizing the functional aspect of their product. The physical aspect is sometimes represented too, for example when the architecture represents how the software is deployed on a set of computing resources.

The overview of SysML presented in this paper covers all the diagrams available in SysML. We explore most of the constructs attached to this diagram and refer to the OMG specification [OMGSysML] for the ones that we do not address. In Figure 1 we compare SysML1.0 to UML2.0 in term of re-use. The text in the figure summarizes the various diagrams available in SysML. Requirements, Parametrics and Allocations are new diagrams available only in SysML. Activity and Block diagrams are reused from UML2.0 and extended in SysML. Lastly, State Machines, Interactions and Use cases are reused from UML2.0 without modifications.

We explore the capabilities of SysML through an example: the *Rain Sensing Wiper (RSW)* system. This sample problem is inspired from a real-life product failure that can be easily found using your preferred Internet search engine. In Appendix A we explain in details the background story of the example that we are using in this paper. This example is a nice illustration of the importance of having an understanding of a product at the level of its sub-systems in order to prevent complex failure modes involving costly product recalls. In the story, the product manufacturer endures a lengthy (hence costly) root cause analysis that eventually requires a design change. In this article, we present a model that is resilient to the failure experienced by the manufacturer.

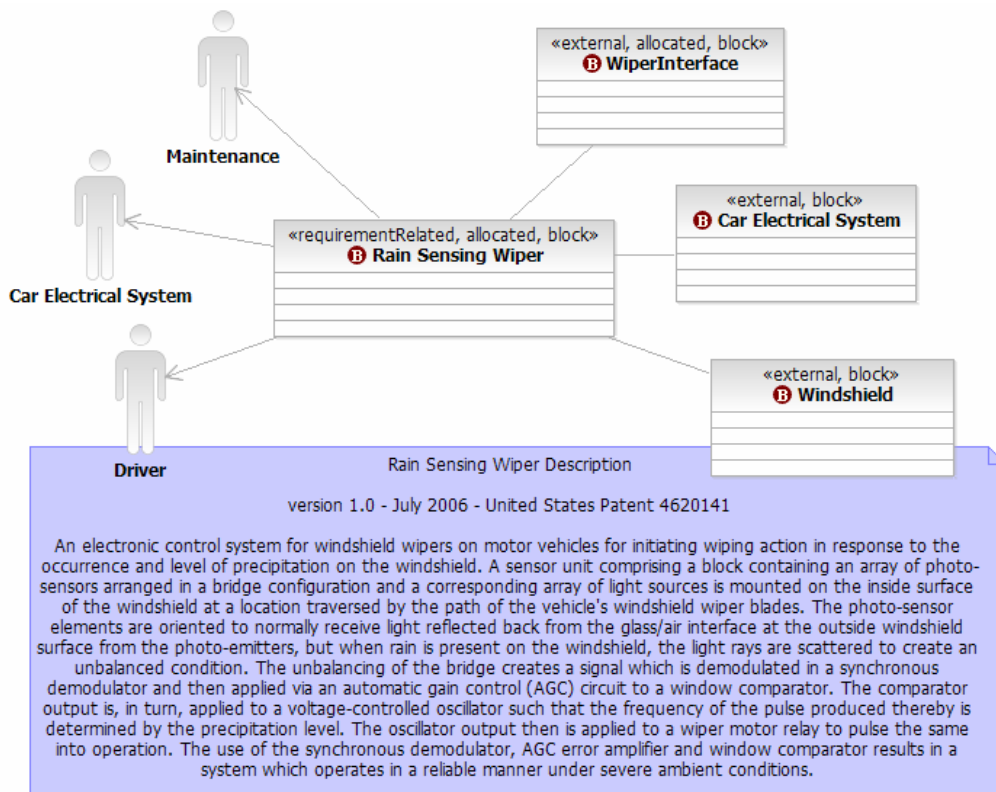
We briefly explain now the purpose of the RSW: The goal of the RSW is to wipe the surface of the windshield automatically (i.e. without user intervention) whenever droplets of liquid are detected. In addition, the amount of liquid detected dictates the speed of the wiper. This system has three main components: the software that controls the behavior of the wiper, an electronic control unit that executes the software and a sensor fixed on the windshield whose task is to sense droplets through the windshield.

In this article, we give a detailed model that describes many aspects of this system. This example constitutes a realistic product in the area of embedded electronics (for the automotive industry) whose design greatly benefits from a SysML representation. We conclude this paper by summarizing the capabilities offered by SysML and give a perspective for the potential improvements that SysML brings for products and systems development.

## 2. Context, Requirement and Use Cases

When modeling a system, an important primary task is to decide what belongs to the system and what does not. The *Context Diagram* is an informal means (informal in the sense that it does not carry precise semantics) to represent the boundaries of the system. In Figure 2 we show a context diagram for the RSW. Three actors for the system are identified in the context: *Maintenance* (for repair purposes), *Car Electrical System* (to activate the system in the car) and *Driver* (to manually disable the system for example.) Three external systems are considered here: the wiper interface, the windshield and the car electrical system. Note that the car electrical system also provides electrical power to the RSW. Hence it is considered

both an actor and external system. The context diagram establishes the scope of the system. Note that a user-defined keyword “external” is used to qualify the external components.



**Figure 2 Context diagram for the Rain Sensing Wiper system.**

In the introduction of this paper, we explain that the conceptual stage of the lifecycle succeeds to the analysis of the customer needs into product requirements. Requirements have been traditionally represented as text (accompanied with figures and drawings) and stored in files or databases. The requirements describe all the product functions and the constraints under which these functions should be realized.

SysML allows the representation of requirements as model elements. Hence requirements become an integral part of the product architecture. The language offers a flexible means to represent text-based requirements of any nature (e.g. functional or non-functional) as well as the relationships between them.

In Figure 3 we represent a requirement diagram for the RSW. Note that it contains both functional and non-functional requirements. Requirements in SysML are abstract classifiers (i.e. they cannot be instantiated) with no operations or attributes. They cannot participate in associations or generalizations, however a set of predefined relationships help to characterize the relationships between the requirements. We review these relationships below.

Sub-requirements are related to their parent requirement using the crosshair relationship (that denotes namespace embedding). For example in Figure 3, some of the sub-requirements of the requirement *Automatic Wiping* are connected to it using the crosshair. The parent requirement is a package for the embedded requirements. In

that sense, deleting the parent requirement will automatically delete all the embedded ones. Another example of requirement acting as a package for other requirements is the requirement *Core Functions* which contains two sub-requirements. For readability in the model, a user-defined keyword “package” is rendered next to the *Requirement* stereotype.

During requirement analysis (e.g. decomposition and flow-down) new requirements are created by derivation. These new requirements can be connected to the initial ones with the *DeriveRqt* dependency. For example in Figure 3, a set of core functions for the product are derived from the set of requirements under *Automatic Wiping*. The name *DeriveRqt* was chosen in order to avoid any confusion with the standard *Derive* dependency in UML2.0. Other examples of derived requirements are the technical choices for each function (see the box *Technical Choices* in Figure 3.) Note that in the Figure, the designer captures a *Rationale* comment to explain his choice for using a sensor fixed on the windshield. A last example of derived requirement is the quality requirement *System Calibration* stating that the system should be calibrated. This is the requirement added to the product after the infamous RSW failure was identified (see the Appendix A for more details.) The satisfaction of this requirement insures that the product will be resilient to changes in the sensor and windshield characteristics.

Another relationship between requirements is *Refine*. An example of requirement refinement is shown in Figure 3. The requirement on speed actuation is refined by the possible selection for speed (slow, medium or fast.) Lastly, a generic *Trace* dependency can be used to emphasize that a pair of requirements are related in some way or another. In Figure 3, the requirement for manual deactivation is traced to the one about automatic deactivation.

Requirements have a number of derived attributes to store the status of the relationships reviewed in the above paragraphs. We will see later in this paper that these attributes become particularly handy when requirement relationships are represented outside requirements diagrams.

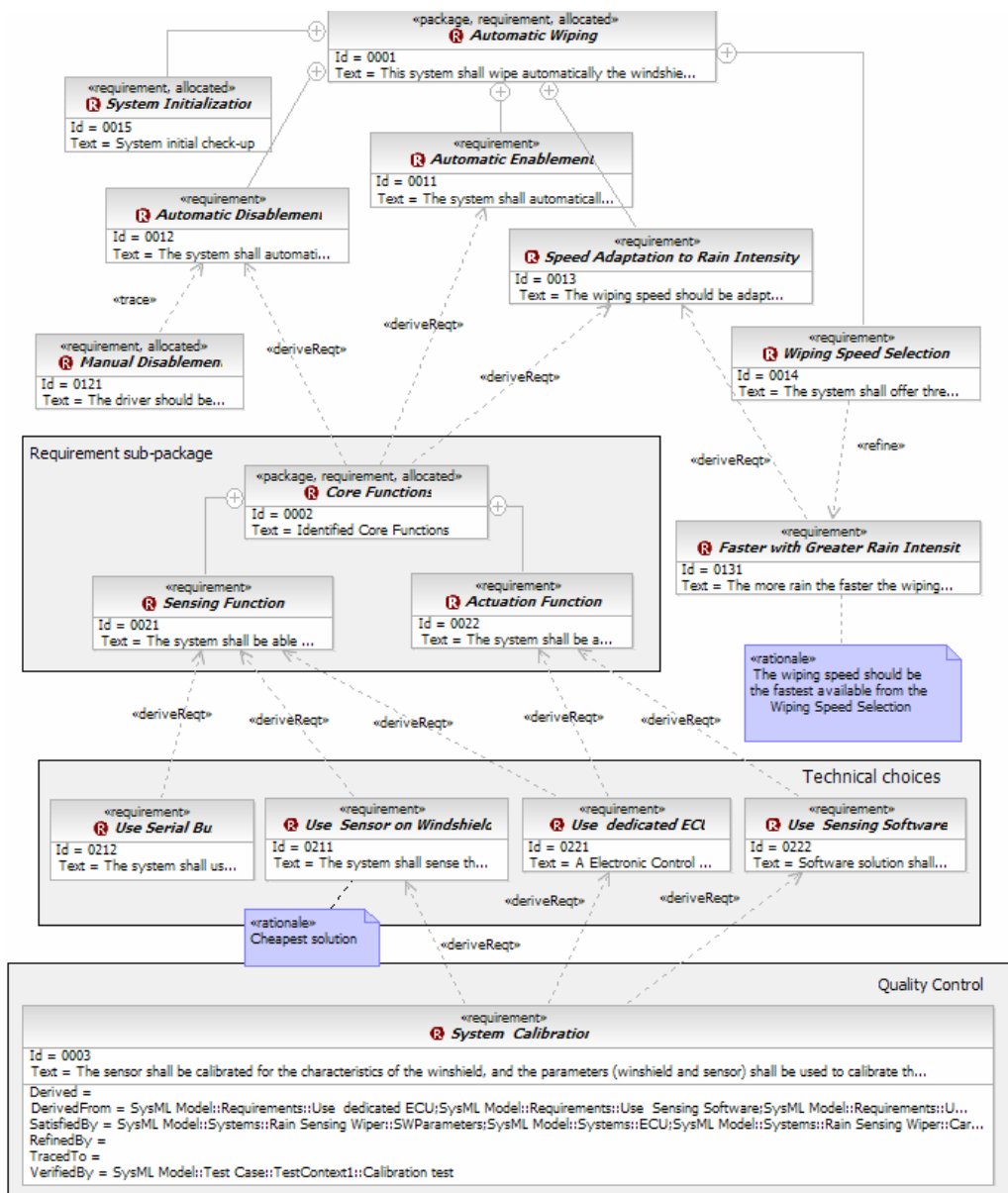
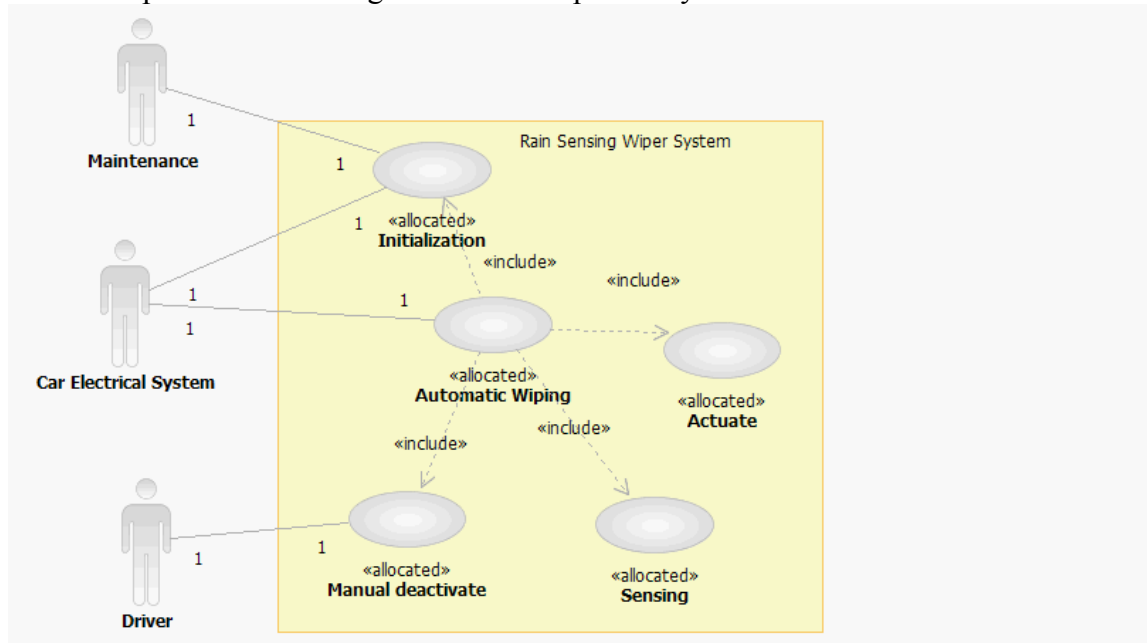


Figure 3 SysML Requirement Diagram for the Rain Sensing Wiper system.

Often requirements are elicited during the whole product lifecycle and additional requirement diagrams are used to represent them. Hence the product requirements are typically laid out on a set of requirement diagrams.

SysML provides a generic model for requirements that allows the modeling of both functional and non-functional requirements. A non-normative set of requirement types are proposed in the appendix of the OMG SysML specification [OMGSysML]. Specific types of requirements (for example related to timing or scheduling, etc) are handled by language extensions. SysML supports a profile mechanism to extend the language. The Object Management Group (OMG) has released a series of modeling standards that address specific needs: for the modeling of non-functional requirements related to performance and quality [QoS, STP], and for the modeling of test cases [Testing profile]. These profiles can be used in SysML without restriction.

SysML provides a use case diagram that is inherited from UML2.0 without modifications. In Figure 4 we show the interaction of the external actors with some of the main use cases (represented by ellipses) owned by the RSW. We represent the three actors and connect them to their respective use cases. In this figure, a central use case *Automatic Wiping* is composed of a series of sub-use cases. The hierarchical relationship is modeled using the *Include* dependency.



**Figure 4 SysML Use Case Diagram**

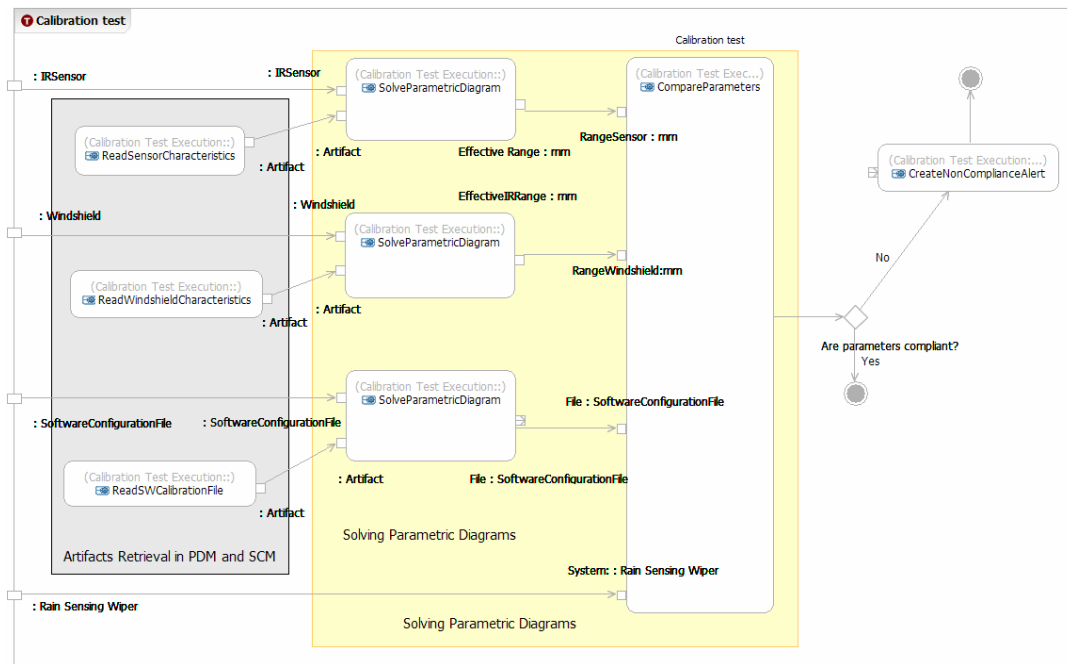
SysML has the capability for representing test cases and attach them to their related requirements or use cases. A test case can be an operation or a behavioral model (Interaction, State Machine or Activity.)

In Figure 5 we show a test case for the RSW. The test case verifies the requirement *System Calibration* (see Figure 3.) This is done by: First, retrieving the characteristics of the different components (sensor, windshield and software configuration file.) Second, using these characteristics to compute an operating range (both for the sensor and windshield) in order to assess their compatibility. If the sensor and windshield are compatible, then the test case is successful. Otherwise, an alert is triggered. The actions in the activity diagram contributing to each step are enclosed in illustrative boxes (Figure 4a) for the sake of clarity.

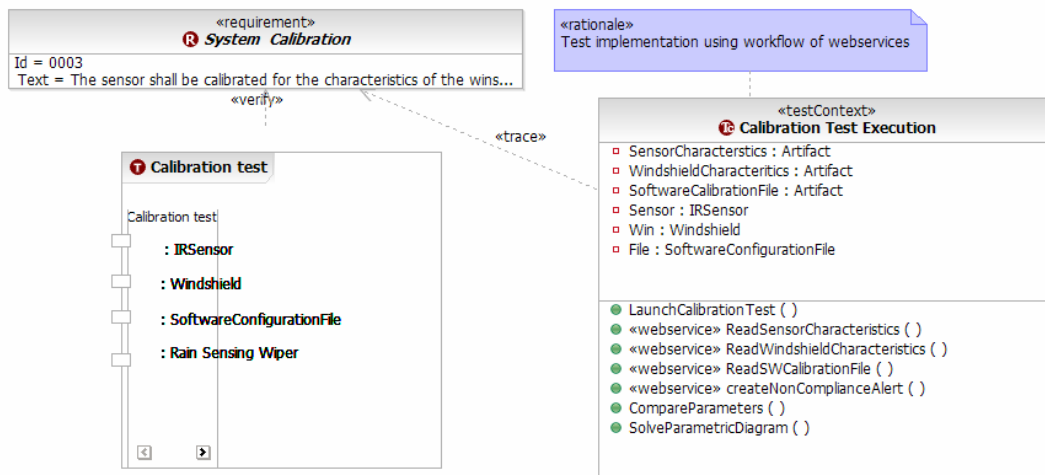
The first step is realized in this example using a set of webservices to access the repositories containing the artifacts related to the different components (see the leftmost enclosing box in Figure 4a.) More precisely, the bill of material (e.g. in a product data management system) is queried for the characteristics of the sensor and the windshield and the configuration file is retrieved, for example, from a software configuration management system.

The second step (see central box in Figure 4a) is realized by defining constraints on the attributes of the sensor and the windshield in SysML Parametric Diagrams. We will explore how these diagrams are constructed later in this paper.

In Figure 4b a test context is created to host prototypes for the webservices and other functions necessary to execute the test case. This context is traced to the requirement. The activity diagram uses the functions of the test context for its execution.



(a) The test case is realized with an activity diagram.



(b) Requirement and test case traceability

Figure 5 SysML Test Case for the quality requirement *System Calibration*.

This concludes our review of the capabilities of SysML for modeling uses cases and requirement. In the next section, we show how SysML is used to create a product structure that satisfies these requirements.

### 3. Structure of the Rain Sensing Wiper system



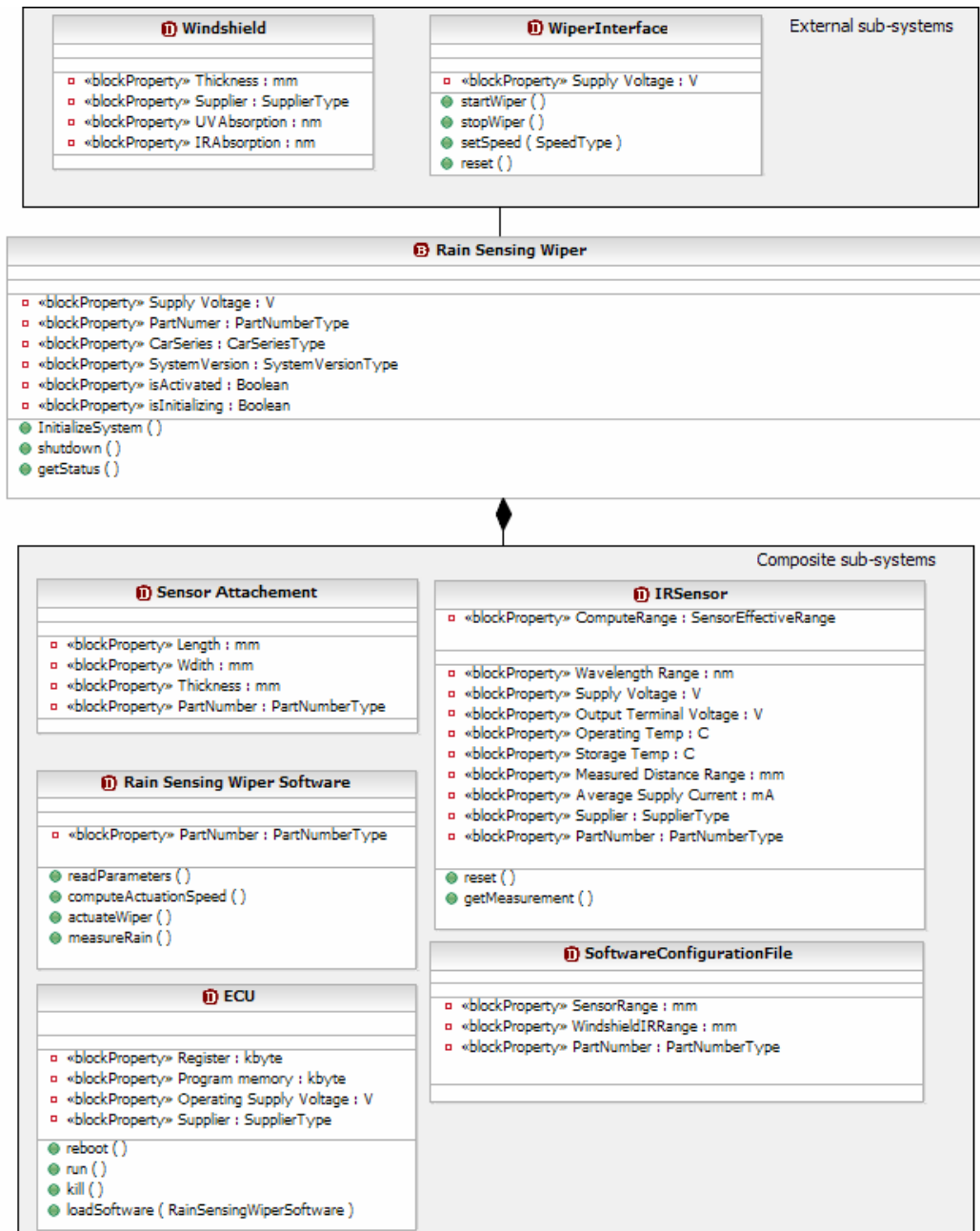
In this section, we build now a structure for the RSW. We assume that a set of sub-systems and components have been identified through the requirements engineering process based on stakeholder concerns such as cost, performances, etc.

SysML provides a basic structural element called *Block* whose aim is to provide a discipline-agnostic building block for systems. Blocks can be used to represent any type of components of the system, e.g. functional, physical, and human, etc. Blocks assemble to form architectures that represent how different elements in the system co-exist.

The *SysML Block Definition Diagram* (BDD) is the simplest way to describe the structure of the system. It is the equivalent to the Class Diagram in UML. It is used to represent the system decomposition using for example associations and composition relationships. The BDD is ideal to display the features of a block, such as its properties, and operations. SysML allows blocks to own special types of properties: *Block Properties* and *Distributed Property*. Block properties impose additional constraints on classic UML Properties, and can for instance own a SysML *Value Type*. Value Types are designed to hold units (e.g. physical units) and dimensions. Distributed Properties let the user apply a probability distribution to the values of the property. SysML proposes model libraries for possible values of units, dimensions and probability distributions.

In Figure 5 we show a BDD for the RSW. For the sake of readability of the diagram, we do not render the associations between the sub-systems and the *Rain Sensing Wiper* element, although these associations exist in the model. Instead we use an illustrative box around each set of components (composite and external) and a black diamond shape over the composite component as a visual clue for composition. The main components of the RSW are: an interface to actuate the wiper, an electronic control unit, a sensor and the windshield element. Both the interface and the windshield can exist in the car with or without the RSW (In SysML they are so-called *reference properties*.)

The properties and the operations for each block are visible in Figure 5. Properties (more precisely SysML BlockProperties, shown using the stereotype <<blockProperty>>) are used to model the physical characteristics of the components. The operations (called sometimes services) represent the functional aspects of the system.



**Figure 6 Block Definition Diagram for the Rain Sensing Wiper.**

We now examine how the product structure and the product requirements can be related: One of the important consequence of having requirements as model elements is that it allows the designer to specify which components in the system satisfy a given set of requirements. This is called *allocation process*. We show an example of requirement allocation in Figure 7. In the figure, the part on the left hand side represents some elements of the RSW, and the part on the right hand side is a hierarchy of requirements. One way to perform allocation is to use the *Satisfy* dependency. In the figure the rain sensing wiper model element is allocated to the requirement named “Automatic Wiping”. Any element in SysML can be used to satisfy a requirement.

Another way to display allocation is to use a dedicated compartment named “Requirement related”. This compartment displays the status of a set of derived

properties related to requirements. In Figure 7 the element ECU displays this compartment: The ECU element is allocated to the requirement named “Use dedicated ECU”.

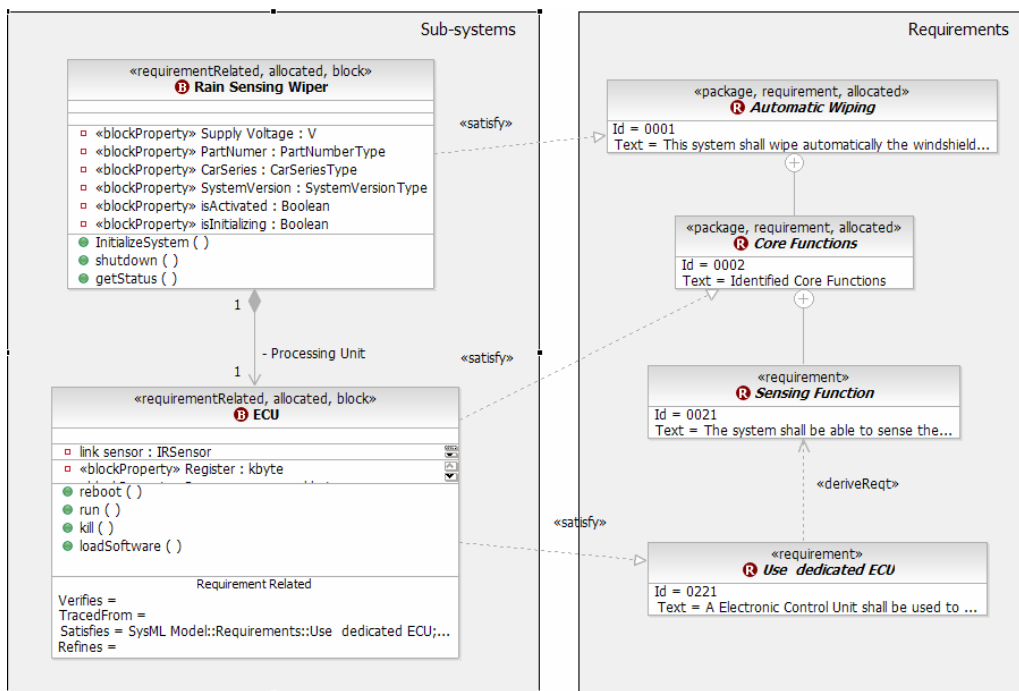


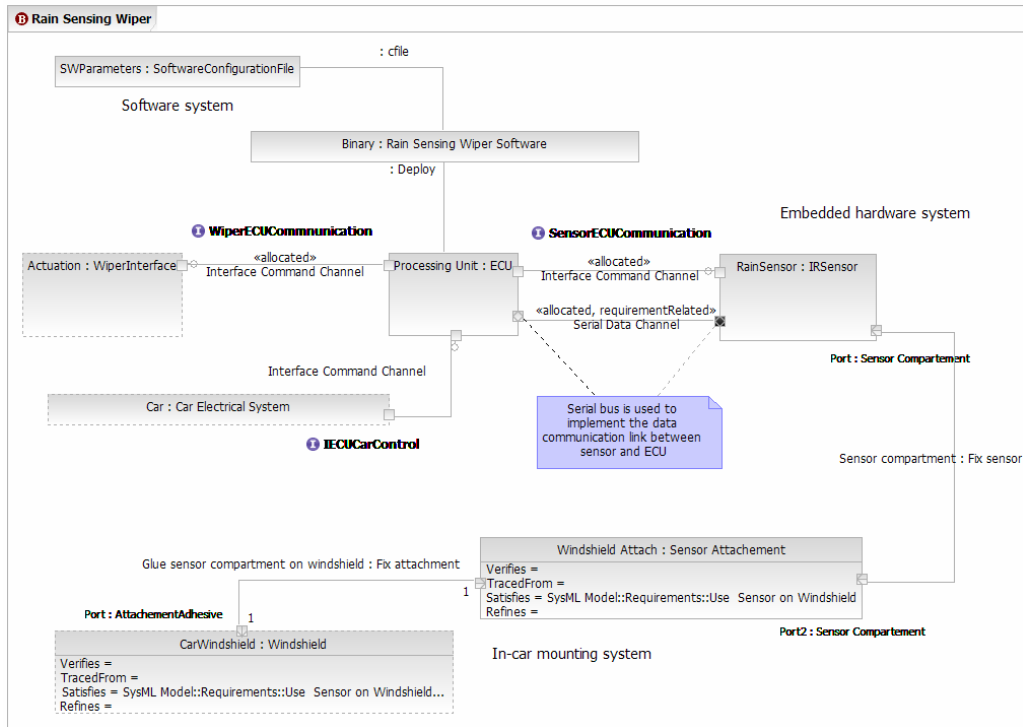
Figure 7 Example of requirement allocation.

The *SysML Internal Block Diagram* (IBD) allows the designer to refine the structural aspect of the model. The IBD is the equivalent of the composite structure in UML. In the IBD properties (or parts) are assembled to define how they collaborate to realize the behavior of the block. A part represents the usage of another other block.

The most important aspect of the IBD is that it allows the designer to refine the definition of the interaction between the usages of blocks by defining *Ports*, as explained below.

Ports are parts available for connection from the outside of the owing block. Ports are typed by interfaces or blocks that define what can be exchanged through them. Ports are connected using *connectors* that represent the use of an association in the IBD.

Two types of ports are available in SysML: *Standard ports* handle the requests and invocations of services (i.e. function calls) with other block, and *Flow ports* let blocks exchange flows of information or material. For standard ports, an *interface* class is used to list the services offered by the block. For flow ports, a *Flow Specification* is created to list the type of data that can flow through the port. When only a single type of object can flow through a port, then the type is used as type for the port directly. Such a port is named *Atomic Port*. The class *Item Flow* is used to represent what does actually flow between blocks in a particular usage context. We refer the interested reader to the standard specification [OMGSysML] for more details on item flows. An example of IBD is given in Figure 8.



**Figure 8 Internal structure of the Rain Sensing Wiper system.**

In Figure 8 we refine our initial description of the RSW by showing how parts are interacting inside the block named Rain Sensing Wiper. Previously to constructing the IBD, we need to define a model for the associations characterizing the relationships between the different blocks. Also additional blocks are defined for example to type the ports. We show this model in another BDD that can be found in Figure 18 (Appendix B.)

The central part of Figure 8 consists of the parts of the system that represent the embedded hardware. The parts underneath are used for mounting the system in the car. The ones above represent the software. A set of standard ports and interfaces are defined to represent the functional aspect of the communication between the parts. For example, the processing unit accesses the actuation interface of the wiper through the interface *WiperECUCommunication*. Details about the interfaces used in this IBD are found in Figure 17.

The processing unit communicates with the sensor using a flow port. The data exchanged is two bitstreams, one containing the measurements from the sensor and another containing synchronization data. The port is typed with a specification of these flows using the element *SensorECUCommunication* (see Figure 17.) Note the direction of the flows in the definition.

For convenience a flow port can be conjugated in the sense that its input and outputs are inversed (flows declared as “in” becomes “out” and vice-versa) with respect to the definition of the interface. This is useful when connecting two systems whose flow ports are conjugated with respect to each other. This is the case for instance between the processing unit and the sensor in Figure 8. A conjugated flow port is represented in black. Since the synchronization data flow is declared as “inout”, the conjugation of the port has no effect on it.

Note that in Figure 8 connectors between ports link parts defined within the block. SysML actually allows direct connection between ports defined at different levels of granularity, for example between a port and another one defined *inside* a part. This type of connector are called *nested connectors*. We refer readers to the standard specification [OMGSysML] for more details about these connectors.

Flow ports are also useful to define physical contact between parts: For example the *SensorAttachment* unit is fixed to the windshield using an adhesive. The block representing the adhesive material (*AttachmentAdhesive* in Figure 18) is used to type the flow port connecting these parts.

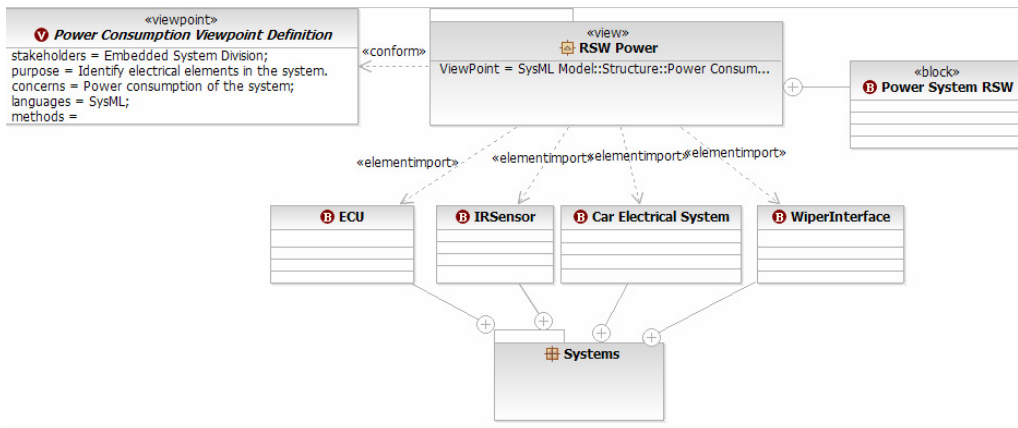
We explain now how to represent requirements allocation in an IBD: Requirements are classifiers and therefore cannot be represented on an IBD. For this type of diagrams, the compartment notation as introduced in Figure 7 is used.

An example is shown in Figure 8: The parts representing the windshield and the sensor attachment are used to satisfy the requirements named “*Use Sensor on Windshield*” and “*System Calibration*”, respectively (see Figure 2.) The satisfaction of these requirements is displayed in each part.

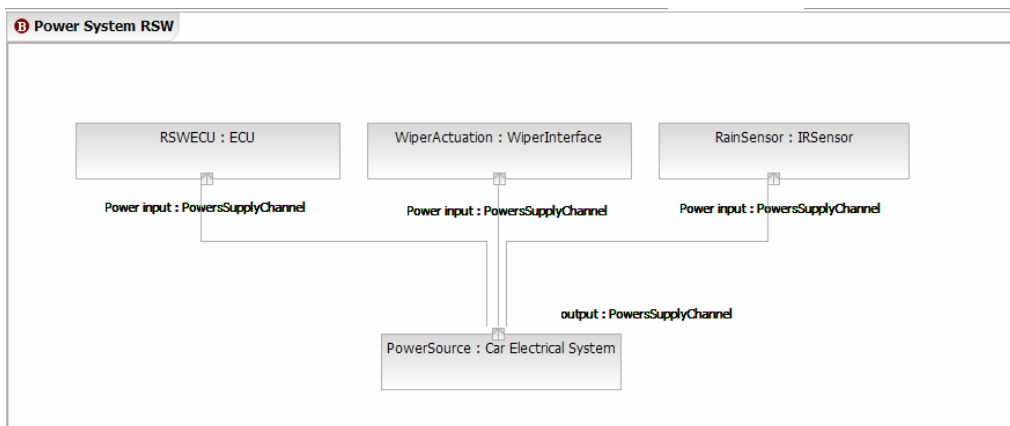
A large and complex model is composed of hundreds, maybe of thousands elements. Hence such a model is laid out over a set of BDD’s. Typically, the content is organized according to stakeholders concerns.

Most design methodologies advocate the use of viewpoints to organize the model, for example according to stakeholder’s interests. SysML provides a model element *Viewpoint* that allows users to capture the characteristics of a viewpoint (for example, targeted stakeholders, concerns addressed, construction rules, etc.) A container element called *View* is then used to organize the model according to the viewpoint description.

Figure 9(a) represents a model for the definition of a viewpoint. In our example model, the elements of the system are contained in a central package called *Systems*. Some elements are imported from this package into a view called *RSW Power* whose purpose is to gather elements playing a role in the power consumption of the system. The view conforms to the definition given by the viewpoint description. Within the view, an element *Power System RSW* is defined to describe how the various imported elements are collaborating in the scope of the power consumption of the system.



(a) Definition of the view RSW Power. The block Power System RSW is defined in the context of the view and uses a set of elements imported from the package Systems for its definition.



(b) An internal block diagram for the block Power System RSW. The diagram describes the roles of the imported elements in the context of the power subsystem of the car.

**Figure 9 Separation of concerns using viewpoint modeling.**

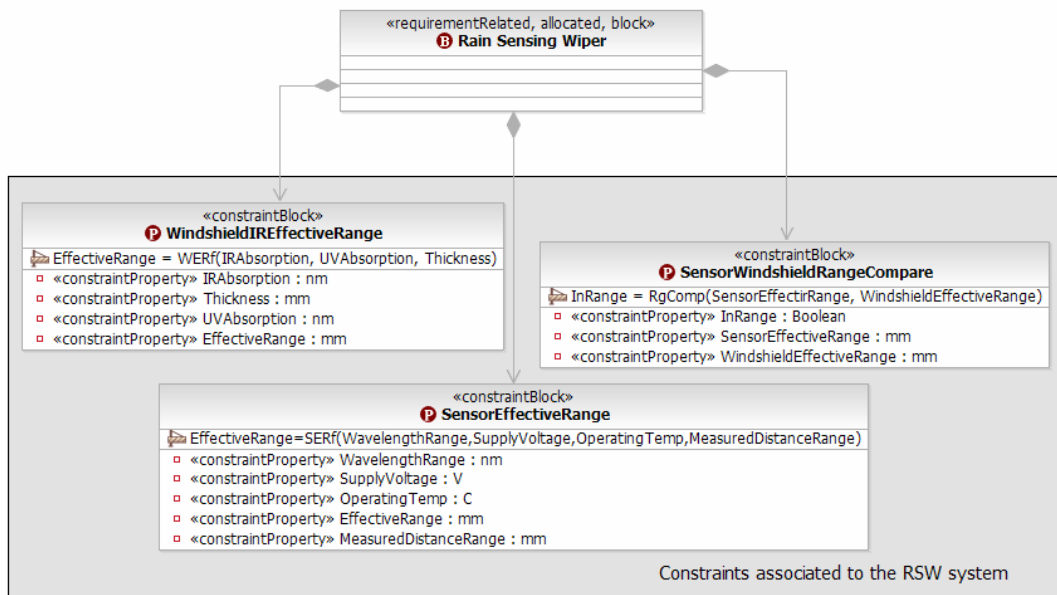
In Figure 9(b) the car electrical system powers the parts using atomic flow ports typed using the *Power Supply Channel* Block (Figure 18). In this case, the direction of the port (*in* or *out*) is specified in one of the port’s attributes.

We have seen so far how attributes are defined for blocks in order to represent their physical characteristics. Often attributes of a set of systems are not independent. Consider two sub-systems A and B having attributes a and b, respectively, and that the constraint {A.a greater than B.b} must hold true. SysML *ConstraintBlocks* allows the engineer to define any relationships (e.g. analytical) between the system attributes. These constraints form networks of expressions that are typically leveraged in simulations, for example for requirements verification. Note that constraint blocks are not instantiated as runtime objects, but rather used to type special properties of blocks, as explained below.

Constraints are properties in sub-systems (i.e. blocks) named *ConstraintProperty* and are typed by *ConstaintBlocks*. A constraint block defines an expression and the attributes that represent its parameters. SysML does not prescribe

any language to represent the expressions or provide a solver for it. This setting is typically offered within the usage of a particular tool.

The RSW uses a set of analytical constraints to verify that the system is properly calibrated (requirement “*System Calibration*” in Figure 2.) Three constraints are shown in Figure 10: The constraint *SensorEffectiveRange* computes an operational range for the sensor, based on some of its parameters. Similarly, the constraint *WindshieldIREffectiveRange* computes an operating range for infrared sensor that can be compared with the one computed for the sensor. Finally the constraint *SensorWindshieldRangeCompare* is used to compare the above values.



**Figure 10 Definition of Constraint Blocks for the Rain Sensing Wiper system.**

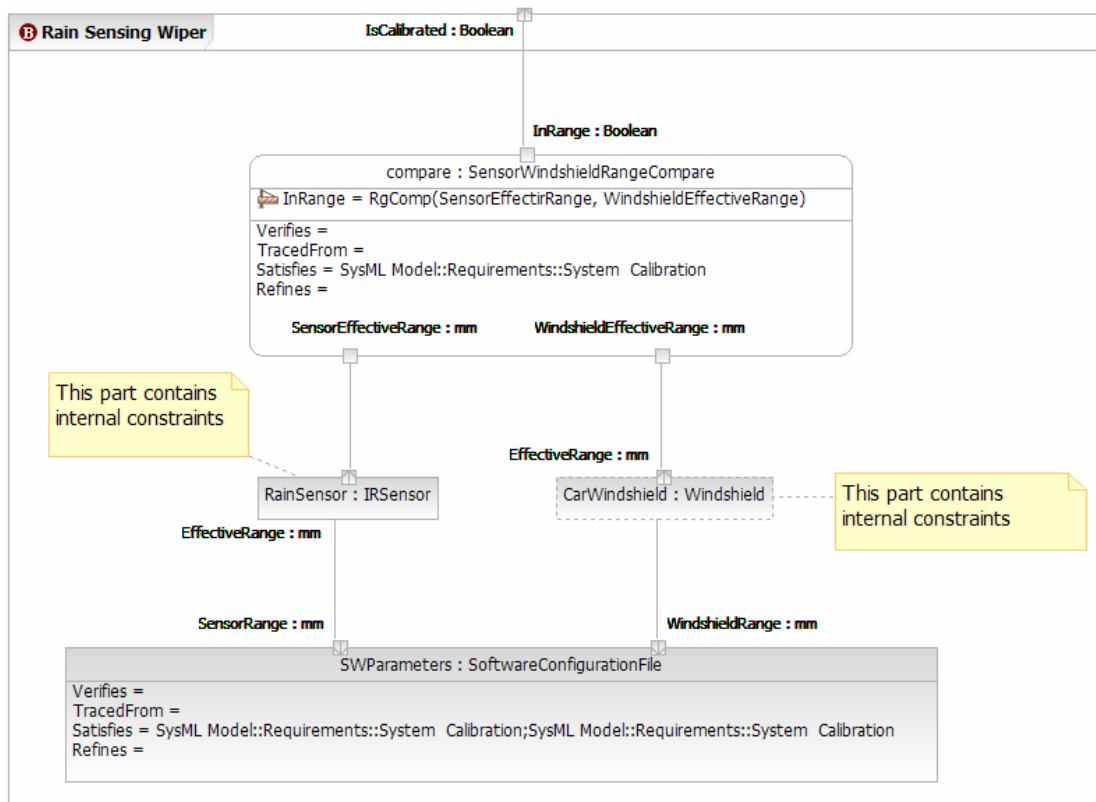
The *SysML Parametric Diagram (PD)* is used to represent the usage of constraint blocks as *constraint properties*. Syntactically the PD is actually similar to IBD. In a PD, constraint properties are connected to each other through the parameters defined by their constraint block. In turn they connect to other properties in the context of their owning block. These other properties must be directly bound to parameters of the constraint properties because they can only play a “feeding role” to the constraints parameters in a PD.

An example of PD is shown in Figure 11. Constraint properties are represented by boxes with rounded corners. In this diagram, both the sensor and windshield parts compute an operational range that is compared by the property named “*compare*”. These values are also fed to the part representing the configuration file (bottom of the figure.) If the sensor and the windshield are compatible, the flag *IsCalibrated* (exposed as a port) is set to true. The verification of the calibration requirement is hence reduced to testing the value of this port. The system is therefore resilient to changes in windshield and sensor characteristics.

The usage of the constraint blocks *WindshieldIREffectiveRange* and *SensorEffectiveRange* can be seen in the diagrams of Figure 19 and Figure 20, respectively (Appendix B.) They are nested in the parts named *RainSensor* and

*CarWindshield* (see comments in the figure.) Note that the parametric diagrams of Figure 11, Figure 19 and Figure 20 are used to implement the second step of test case that we presented in Figure 5.

An attractive aspect of constraint blocks is that they provide a reusable mechanism to define types of constraints. Hence the same constraint can be used several times in the model. It is important to note that a constraint does not specify which variable is an input or an output. Values are assigned by the context and a numerical solver will provide results for the variables of the system. Note to conclude our review of SysML constraints that remarkable work on this topic is available from Peak et al. [GTech].



**Figure 11 Parametric Diagram for the Rain Sensing Wiper system.**

Requirement allocation is shown in PD's using compartments: In Figure 11, the requirement allocation compartment is displayed in both the constraint used for comparison and the part representing the configuration file. These elements satisfy the requirement named *System Calibration*.

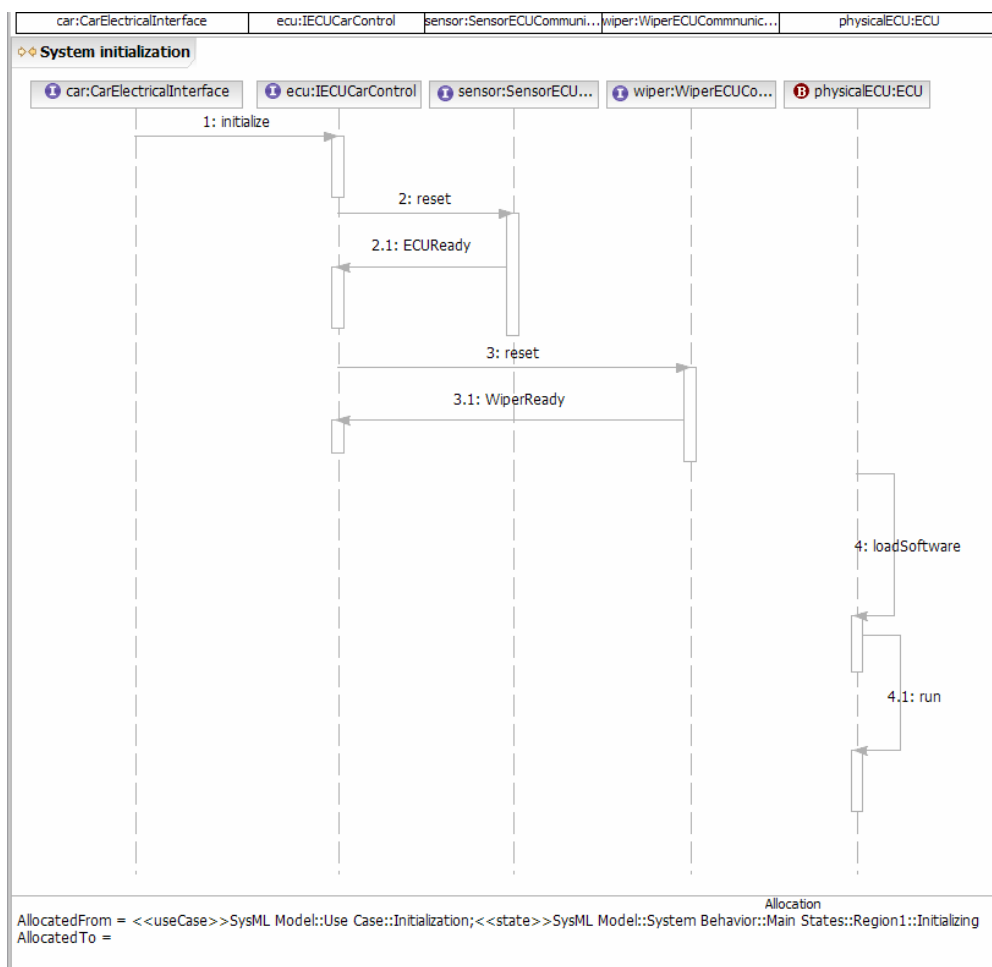
#### 4. Behavior

In this section, we explore how SysML is used to model the behavior of the product. The expression of the behavior of a system equates to realizing its Use Cases under a specified set of non-functional constraints. SysML offers three types of behavioral constructs: *Interactions*, *State Machine* and *Activities*. Several behavioral models from UML are not reused for the sake of simplicity or because of some maturity concerns. We examine these three models below through our RSW example.



The first behavior model that we review is the *SysML Interaction Diagram*. This diagram allows the designer to model a sequence of service calls between components. SysML leverages the UML2.0 interaction model but restricts its use in the interaction diagram only. Other forms of interaction diagrams (e.g. communication diagram) are not used.

In Figure 12 we represent the initialization sequence of the RSW using an interaction diagram. The initialization sequence consists of a synchronization protocol between the components. This diagram is well adapted to represent this type of behavior. Initially, the car electrical system starts the RSW, which in turn queries the sensor and the wiper interface for an acknowledgement. After that, the software is loaded in the memory of the ECU and its execution is started. Once the software is started, it reads the parameters stored in the calibration file.



**Figure 12 Interaction Diagram for the initialization of the Rain Sensing Wiper system.**

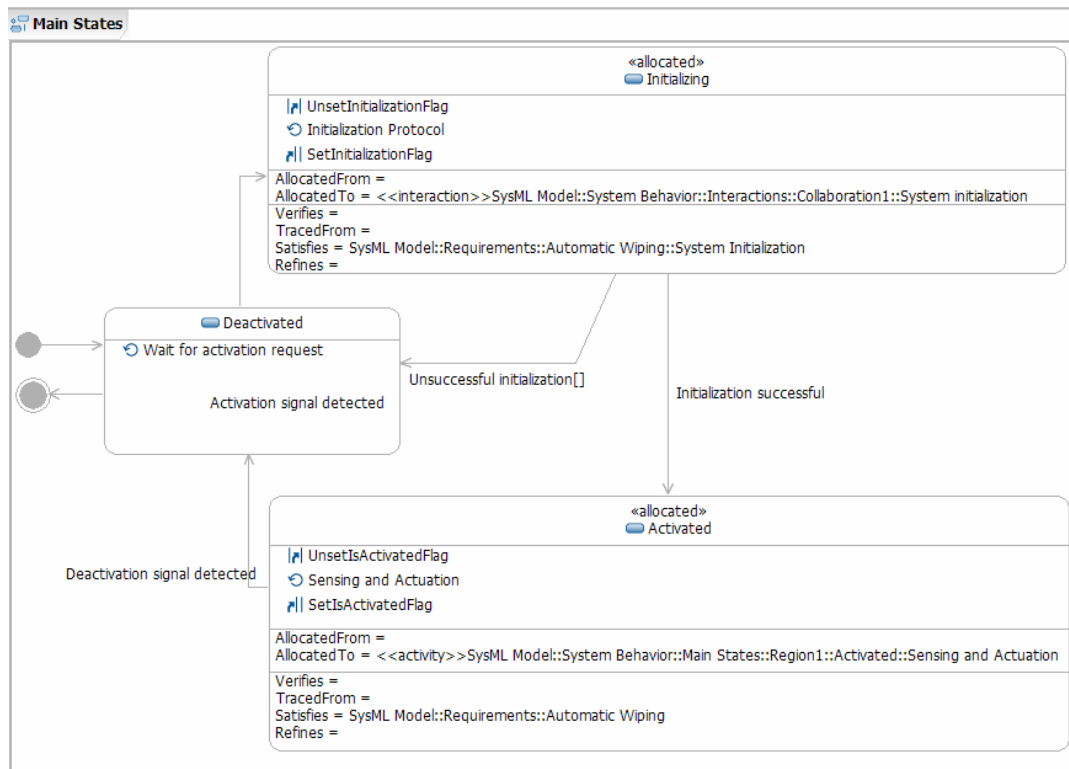
The use case named *Initialization* (Figure 4) is realized by the interaction model in Figure 11. One mechanism for expressing the relationship between the interaction model and the realized use case is to use the *SysML Allocation* mechanism, as introduced below.

In our previous examples, we have seen how requirements are allocated to system elements aiming at satisfying or verifying them. SysML generalizes this concept by allowing any elements to be related to any elements to express a particular

relationship. We will examine this mechanism in more details in the coming examples and in Section [Allocation]. Similarly to requirement relationships, elements in SysML have derived properties to display the status of their allocations to other elements. For example, at the bottom of the interaction diagram, we can read that the behavior model is allocated to the use case *Initialization*. In addition, the model is allocated to the system state named *Initializing* (see below.)

The second behavior model that we introduced is the *SysML State Machine Diagram*. This diagram is used to represent the different states of the RSW. This behavioral model is also reused from UML and not extended in SysML. Protocol State Machines from UML2.0 are excluded from SysML for simplicity.

In Figure 13 the different states of the RSW are represented as well as their transitions. Three states are identified: *Deactivated*, *Initializing* and *Activated*. In deactivated mode (for instance manually by the user), the system waits for an activation command. When this signal is received, it transits to the initializing state. In that state, the interaction sequence in Figure 12 is executed. When completed successfully, the system transits to the “activated” state. Note that, when entering and exiting states, flags are set and unset, respectively. These flags are used for example to display the current status of the system on the driver dashboard.



**Figure 13 State Machine Diagram for the Rain Sensing Wiper system.**

Allocation can also be shown in the state machine diagram as well. In Figure 12 the states *Initializing* and *Activated* both exhibit requirement and allocation compartments.

The third type of the behavior that we review is the *SysML Activity Diagram*. SysML leverages and extends the activity model from UML to support continuous

systems. The *SysML Activity Diagram* offers many innovations presented briefly below. A more thorough description of the SysML activity model is available in [Bock].

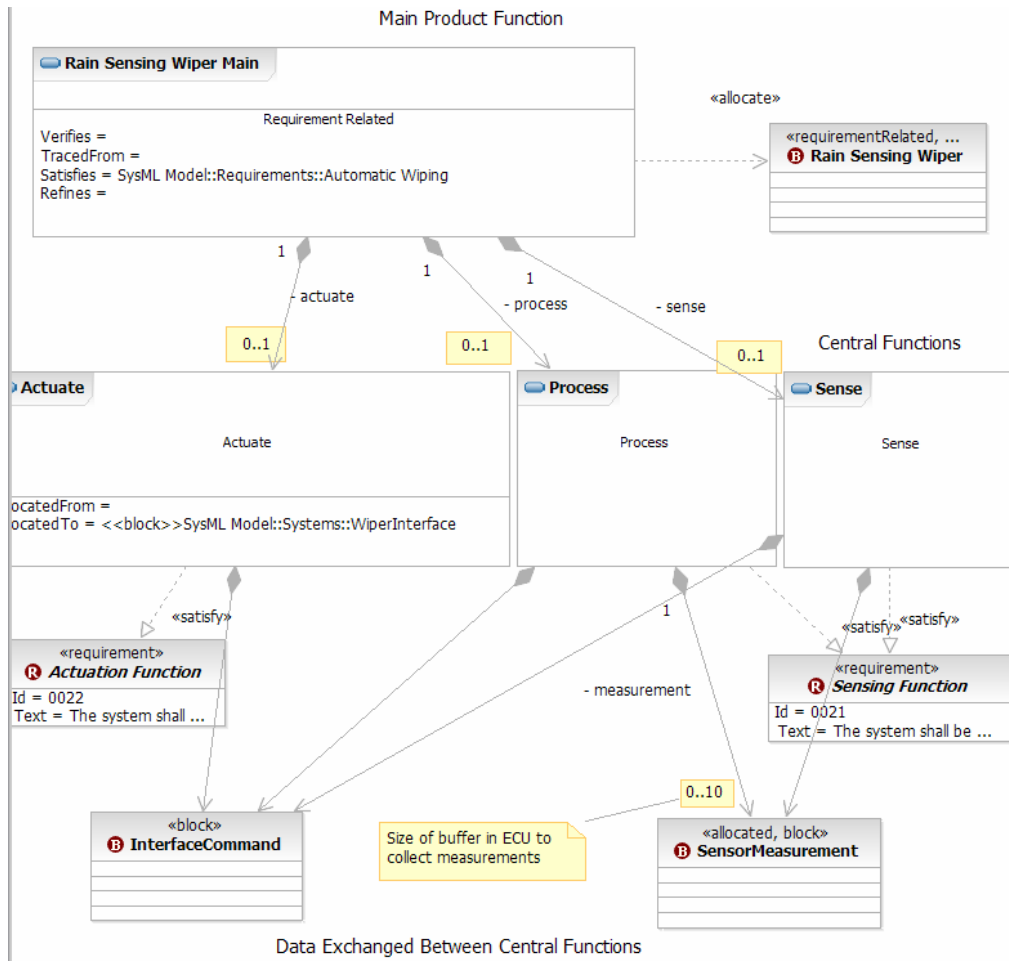
The modeling of activities in SysML consists of describing behavior as a flow graph. An activity is defined as a set of actions represented as graph nodes (these actions are the usage of other activities) linked by edges carrying control flow and data flow between actions. Object nodes represent the type of data that can traverse the flow graph and are used as containers for the data as it traverses the graph. For example, object nodes are used to store input parameters of actions (called pins.) Finally, control nodes are used to route control and data flows through the activity (for example create forks, duplicate flows, etc.) An example of activity model is shown in Figure 15.

Since UML2.0, activities are classifiers and can be represented in Class Diagram. SysML clarifies the semantic of association between activities. As a result, Activities can be related to each other to represent for example functional decomposition in a similar way that blocks represent structural decomposition in a BDD. In Figure 14 we represent a decomposition of the main functions of the system.

In this setting, the upper limit at the part end represents the maximum number of instances that can execute concurrently. In Figure 14 the lower and upper limits for each activity owned by *Rain Sensing Wiper Main* (i.e. *Actuate*, *Process* and *Sense*) are enclosed in an illustrative box. The example setting allows only one instance of each activity to be run concurrently in the system.

Activities can also be associated to classifiers when the latter are used as type of object node. In the case, the upper limit at the classifier end represents the maximum number of instances that can reside in an object node at any given time. In Figure 13 we can see that the upper limit for the number of instances that can be owned by *Process* is ten.

In Figure 13, we can see that the activities *Sense* and *Process* share a common type of object node named *SensorMeasurement*. Also, all activities use the type *InterfaceCommand*. Note that both these blocks are specializations of a data type named ***Bitstream***. The bitstream modeled by the block *InterfaceCommand* is used to implement the services communications between the ECU and the wiper interface (defined by the various service ports between the parts.)



**Figure 14 Activities and object node types for the sensing activity of the RSW.**

The requirements that the Actuate and Sense functions satisfy are also represented in Figure 14. In addition the main function of the product is allocated to the rain sensing wiper block using an *Allocate* relationship. In this case, the designer expresses that the function is owned by this block. The activity *Actuate* displays its allocation compartment which shows that this activity is allocated to the block *WiperInterface*.

The rain sensing function is implemented in the activity diagram in Figure 14. The three functions defined in Figure 13 (Actuate, Process and Sense) are used as actions in the activity.

In Figure 14 the data flows across edges that connect the actions through their pins. The type of data that can flow between actions is defined by the type of their pins. For example, objects of type *InterfaceCommand* are flowing from the action Process to the action Actuate.

SysML introduces notions to specify the rate at which data can flow across edges and parameters (pins) of activities. Two types of the rate are defined: *Discrete* and *Continuous*. In the example of Figure 14, the bitstream exchanged between the function Process and Actuate is limited in rate by the serial link that is used to connect the parts owning these functions (these parts are shown in Figure 8.) In Figure 14, this is modeled by the edge (of type Discrete) named "serial data channel". Edges of type

Continuous are used when the time interval between objects tends towards zero. Note that in order to use rate-controlled edges and pins, activity parameters must be “streaming”<sup>1</sup>. An activity with streaming parameters can accept values at any time during its operation.

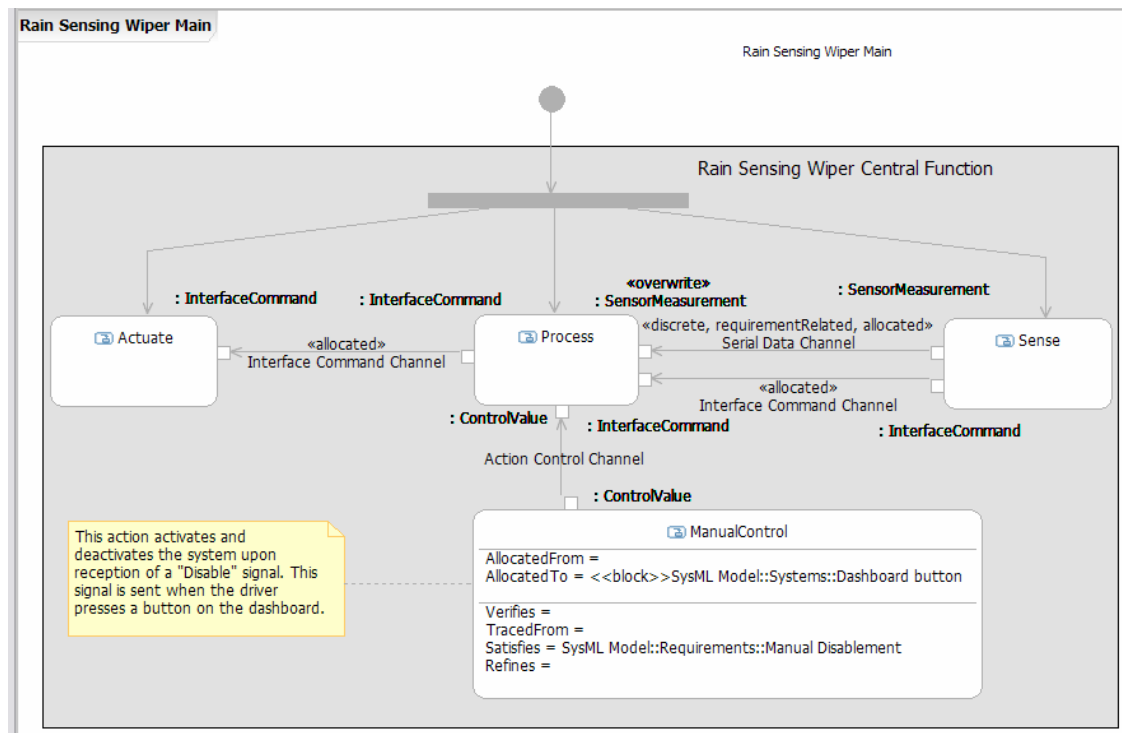


Figure 15 Sensing activity for the Rain Sensing Wiper system.

The UML2.0 action model only enables actions to start. SysML introduces the notion of *control operators* as a special type of actions able to enable or disable other actions. A control operator named *ManualControl* is used in Figure 14 to start and stop the rain sensing function. It deactivates the function upon reception of a “deactivate” signal (for example triggered by the pressure of a button on the dashboard.) It can restart the function later when receiving the proper signal. The type of data exchanged between the actions *Process* and *ManualControl* is of type *ControlValue*. SysML defines some types of control values (*enable action*, *disable action*) that can be extended by users according their needs (for example, a *suspend action* value can be added.)

SysML also introduces two new types of object nodes to support continuous concept such as transient states (e.g. for the modeling of electrical signals.) In Figure 14, the action *Process* exposes a pin of type *Overwrite* to collect the data sent from the sensor. An object node of type *overwrite* will ensure that the latest measurements from the sensor are available to the *Process* action by erasing older ones in the object node. Note that up to ten values can be stored in the pin according to the bound (0..10) defined in Figure 13. The second new type of object node is *NoBuffer*: In this case, the pin does not store any value; hence a value is discarded if it is not directly accepted by outgoing edges (in the case of an output pin) or an action (when used as input.)

<sup>1</sup> UML2.0 notion.

Lastly, SysML supports assigning probabilities to activity edges (whose source is a decision node or object node) and parameter sets. It allows the modeler to assign a probability for an object value in order for it to traverse an edge. In the case of a parameter set, it assigns a probability for the set to be assigned a value at runtime. This feature can be used for instance to simulate loss in a communication channel.

Allocation to requirements or other elements is shown in the activity diagram using the compartment notation. In Figure 14, the action called “manual control” satisfies the requirement named “manual disablement”. Also, this action is allocated to the block representing the dashboard button to emphasize that this button causes the system to deactivate.

## 5. Allocation

The review of the *SysML Allocation* mechanism will conclude this presentation of the Systems Modeling Language. We have introduced some examples of allocations in the previous examples of Figure 11 to 14.

The concept of allocation allows the user to bridge between various modeling techniques. By allowing allocation between any pair of elements, the designer can enforce consistency between the various parts of the model.

Through the examples of Figures 11 to 14, we have seen that allocations can be displayed using special compartments or using the Allocate dependency. The usage of this dependency is only possible when the related elements can be represented on the same diagram.

Allocation is often used to represent mapping of function to structure, as used in the example of Figure 13. Mapping between elements can be complex and require the display of several relationships. To allow a scalable display of dependencies between elements, SysML provides a tabular notation for relationships, as explained below.

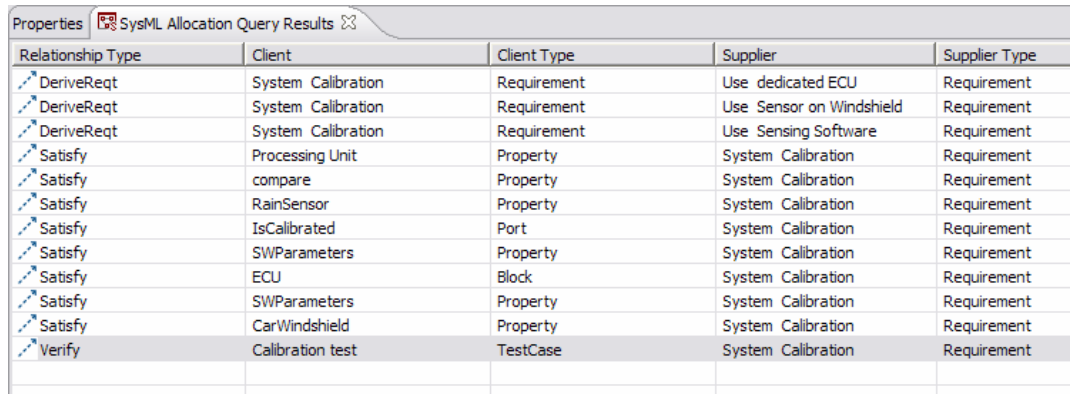
In Figure 15 we show the allocation relationships between the edge named “serial data channel” in Figure 14 (linking the actions Process and Sense) and other elements in the model. Note that the type of an activity edge is shown as *ObjectFlow* in the table. The figure shows that the edge is allocated to four elements in the model.

Relationship Type	Client	Client Type	Supplier	Supplier Type
Allocate	Serial Data Channel	ObjectFlow	Serial attachment	Association
Allocate	Serial Data Channel	ObjectFlow	Serial Data Channel	Connector
Allocate	Serial Data Channel	ObjectFlow	SensorMeasurement	Block
Allocate	Serial Data Channel	ObjectFlow	SensorECUCommunication	FlowSpecification

**Figure 16 Display of allocation relationships using the tabular notation.**

Requirement dependencies can also be displayed using the tabular notation. To conclude this review of the SysML Language, we show in Figure 16 the result of

an allocation query for the requirement *System Calibration*. Note that the direction of the relationship depends on whether the requirement is the client or the supplier.



Relationship Type	Client	Client Type	Supplier	Supplier Type
DeriveReq	System Calibration	Requirement	Use dedicated ECU	Requirement
DeriveReq	System Calibration	Requirement	Use Sensor on Windshield	Requirement
DeriveReq	System Calibration	Requirement	Use Sensing Software	Requirement
Satisfy	Processing Unit	Property	System Calibration	Requirement
Satisfy	compare	Property	System Calibration	Requirement
Satisfy	RainSensor	Property	System Calibration	Requirement
Satisfy	IsCalibrated	Port	System Calibration	Requirement
Satisfy	SWParameters	Property	System Calibration	Requirement
Satisfy	ECU	Block	System Calibration	Requirement
Satisfy	SWParameters	Property	System Calibration	Requirement
Satisfy	CarWindshield	Property	System Calibration	Requirement
Verify	Calibration test	TestCase	System Calibration	Requirement

Figure 17 The tabular notation is used to display the requirements relationships.

## 6. Conclusion

In this paper we have toured the different capabilities that SysML offers to system engineers and product designers. SysML is aimed at supporting the conceptual stage of the lifecycle of the product. This stage is preceded by the decomposition of the customer needs into product features. We have seen that SysML allows the representation of these features as requirements in the model. In turn these requirements can be allocated to the use cases, to the sub-systems and components (whether functional or physical) identified for the product.

The conceptual stage requires the specification of the various sub-systems and the need for details depends on their level of integration. SysML provides a set of constructs to support the description of the structure of the product. Blocks are used to model sub-systems and components, and ports support the description of their interfaces. Dependencies (e.g. analytical) between structural properties are expressed using constraints and represented using the parametric diagram.

In addition to structure, the conceptual stage should clarify how the product behavior is expressed through the interaction of its components. For example, behavior modeling gives a detailed description of the product use cases. SysML provides three means for explicating the product behavior, namely interactions, state machine and activities. These three mechanisms are built as a unified behavior concept and can consequently be orchestrated in a single, uniform and complex behavior model for the whole product.

A complex product model is formed by several “sub”-models of different nature (for example requirements, blocks, constraints, activities, etc.) SysML provides a mechanism to relate different aspects of the model and to enforce traceability across it.

The conceptual stage precedes the detailed elaboration of the components within the different engineering disciplines. The conceptual design plays therefore

many central roles in the product lifecycle and we emphasize below some of the most important ones in our opinion.

The formal description of the product at an early stage of the lifecycle improves the understanding of the product requirements and how they answer the customer needs. The allocation of requirements to the model elements ensures that these needs are covered and provides a rationale for the engineer in charge of fulfilling these requirements. The rationalization of the design is therefore a **communication tool** spanning organizational levels and lifecycle stages. It improves communication across teams, between teams (think of the different engineering disciplines) and between teams and decision makers. It uses a generic language (in the sense that it is not specific to any engineering discipline) that accommodates the incremental detailing of the product representation. That last aspect allows coping with organizational levels. Note that such a formal description is well suited to methodologies.

The SysML model provides an electronic representation of the product that is leveraged as a **decision tool**. Trade-off studies are performed by evaluating functions on the model (cost function, estimation of the integration effort.) At an early stage in the lifecycle, often rough estimations are used, hence the model need not necessarily have a great amount of details in order to be used efficiently. When details are added, or artifacts (for example sub-system simulations) are produced by detailed engineering, the model is used to orchestrate the various simulations and perform requirement verification. Hence the SysML model is an evolving decision tool available throughout the whole lifecycle of the product, and not only at the conceptual stage.

The product model represents abstractions of artifacts that are progressively elaborated throughout the lifecycle. These artifacts are distributed across the engineering disciplines participating to the design. Hence the model forms a traceability scaffold that provides a means to measure the development progress, perform change impact analysis, and manage dependencies between processes and the produced artifacts. The SysML model is hence a **management and integration tool** for the stakeholders.

The role of the system model clearly extends beyond the capabilities that we describe above. We aim at discussing some of the attractive ones in our opinion. Modeling for conceptual design is a young discipline and best practices will grow out of it.

## 7. References

- [OMGSysML] *SysML 1.0 Specification* (ptc/06-05-04), OMG final adopted specification, available at <http://www.omg.sysml.org/>
- [QoS] *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms* (ptc/04-06-01), OMG final adopted specification, available at <http://www.omg.org/docs/ptc/04-06-01.pdf>
- [STP] *UML Profile for Schedulability Performance and Time* (ptc/05-01-02), OMG final adopted specification, available at [www.omg.org/technology/documents/formal/schedulability.htm](http://www.omg.org/technology/documents/formal/schedulability.htm)



[Testing profile] *UML Testing Profile* (ptc/05-07-07), OMG final adopted specification, available at [www.omg.org/technology/documents/formal/schedulability.htm](http://www.omg.org/technology/documents/formal/schedulability.htm)

[GTech] Peak RS, Friedenthal S, Moore A, Burkhart R, Waterbury SC, Bajaj M, Kim I (2005) Experiences Using SysML Parametrics to Represent Constrained Object-based Analysis Templates. 7th NASA-ESA Workshop on Product Data Exchange (PDE): The Workshop for Open Product & System Lifecycle Management (PLM/SLiM), Atlanta. See also <http://www.pslm.gatech.edu/topics/sysml/>

[Bock] Conrad Bock, SysML and UML 2 Support for Activity Modeling, Wiley InterScience (www.interscience.wiley.com) DOI 10.1002/sys.20046

## Appendix A: the Rain Sensing Wiper Story

The development of the first Rain Sensing Wiper illustrates how a classic failure to fully conceptualize a product's physical architecture resulted in the discovery of integration issues at servicing time, thus quickly leading to engineering change requests. The scenario revolves around the initial introduction of the Rain Sensing Wiper (RSW) feature in an automobile manufacturer's vehicle program.

Before examining the reason of the failure of the RSW, let us briefly review the characteristics of the system. The RSW contains mechanical (optical mounting device), electronics (IR sensors and ECU), and software (computer vision algorithm) components that are procured by tier-one suppliers. These components are simply integrated by the manufacturer. The main parameters of the system are: (1) the optical and geometric specifications of the windshield, in particular its thickness and glass optical indexes, and (2) the ranges of operation of the electronic optical sensors. The detection software also has normal ranges of operation relative to these parameters, but in addition relies on data about the actual values of the parameters of the windshield.

The fact that the RSW electronics and software specifications include ranges for the relevant windshield properties is important, because it allows more flexibility on the choice of the windshield itself. This is a critical design choice, the procurement and integration costs associated with the windshield being an order of magnitude greater than that of the RSW. For optimal performance, the actual values of the windshield properties should fall near the center of the normal operating ranges both for the RSW sensors and the software; however, acceptable operation should be guaranteed for the whole range.

From a procurement standpoint, the windshield is simultaneously purchased from different suppliers. Depending on the year of production and where the product is manufactured, suppliers may modify the design of their windshields. Also, one or more changes of suppliers can occur during the production phase.

In the failure scenario, which occurred during the first year of the RSW's introduction, a local windshield supplier provided a component whose characteristics were incompatible with the operating range of the sensor. Unfortunately, no requirement for calibrating the system properly (i.e., for verifying that sensor and windshield are compatible) had been captured for the RSW at that point. Thus cars were sent to customers with a non-functioning wiper system.

Initial diagnostics designated the software as the culprit for the malfunction, since it was difficult for mechanics to test its behavior. The other components (ECU, sensor, and windshield) were functioning normally when tested independently. The failure mode for the RSW resided at the level of its sub-systems, which made it difficult for the manufacturer to discover it. After discovering the root cause, a new requirement was captured to ensure that new systems will be properly calibrated at the production stage.

In our SysML model, this requirement is named *System Calibration* and shown in Figure 3 and Figure 5.

## Appendix B: Additional Diagrams

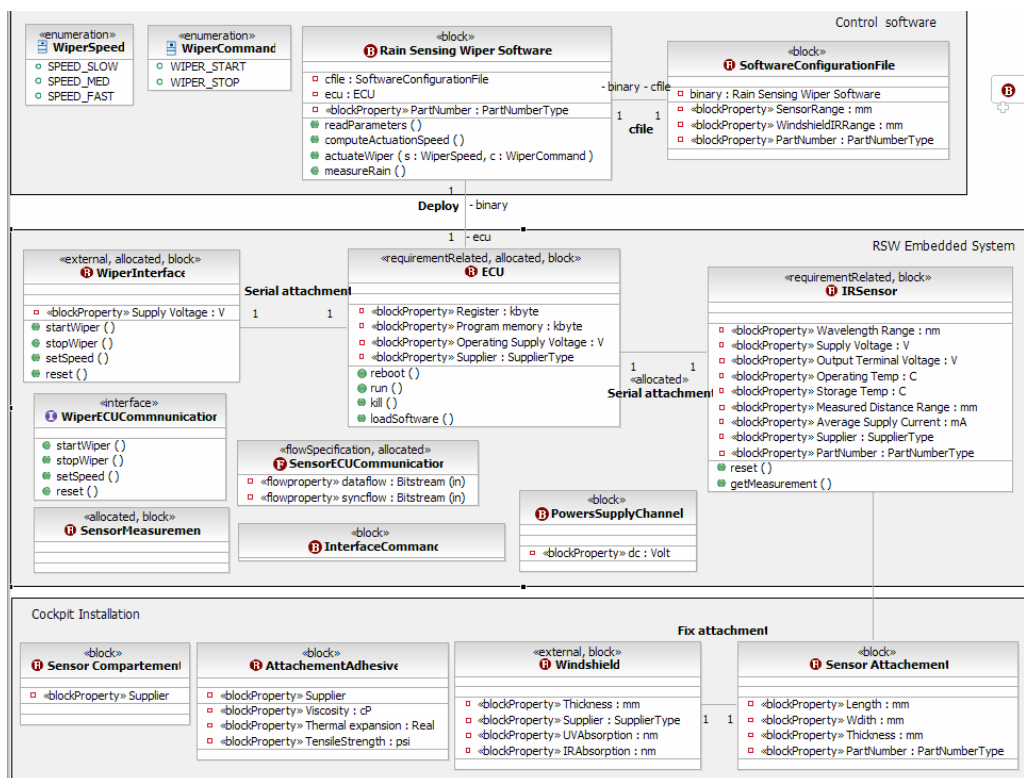


Figure 18 Associations between Blocks and Additional Elements

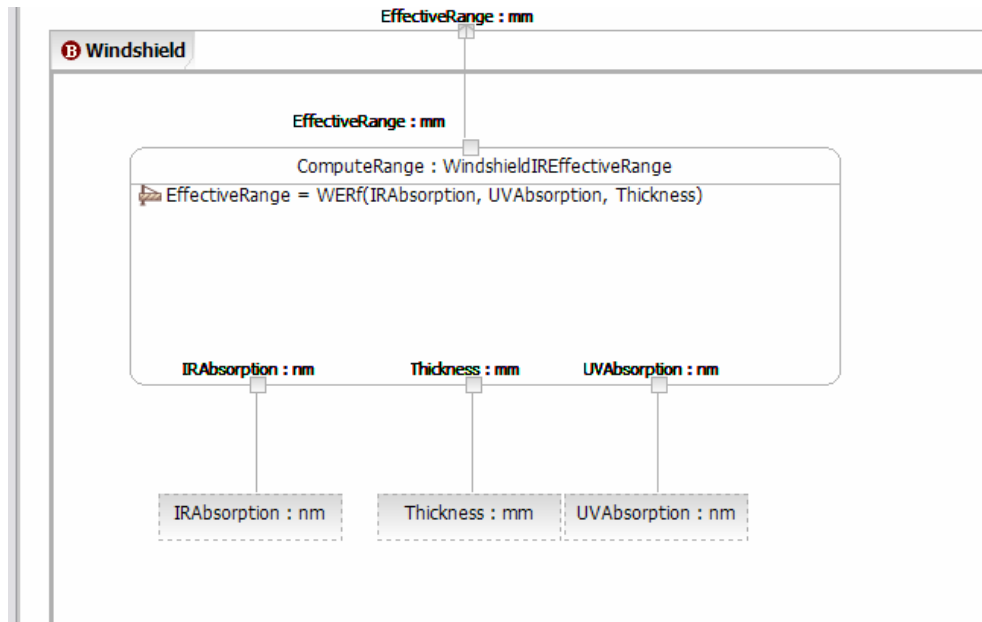


Figure 19 Parametric Diagram for the Sensor.

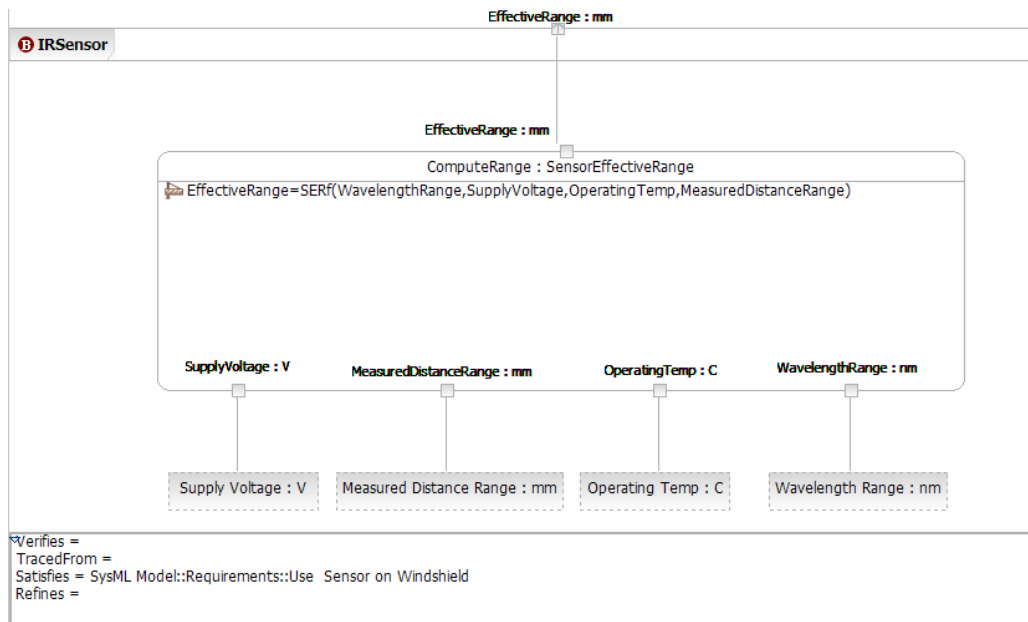


Figure 20 Parametric Diagram for the Windshield.