Regular Paper

Componentization in the Systems Modeling Language

Conrad Bock*

U.S. National Institute of Standards and Technology, 100 Bureau Drive, Stop 8260, Gaithersburg, MD 20899-8260

Received 28 September 2012; Accepted 28 July 2013, after one or more revisions
Published online 20 November 2013 in Wiley Online Library (wileyonlinelibrary.com).
DOI 10.1002/sys.21276

ABSTRACT

This paper describes new capabilities in the Systems Modeling Language that reduce the complexity of specifying systems through componentization, and increase the range of systems that can be specified. Modelers can identify portions of components available for connection to other components, and specify how systems make use of them. This reduces the complexity of specifying system by lowering the number of ways components can be connected, partly by hiding portions of components, and partly by limiting how exposed portions can be connected to others. The paper introduces basic SysML concepts and notations for specifying components and assembling them into a system. It covers capabilities added to SysML enabling a wider range of systems to be modeled, through detailed specification of the portions of components available for connection to other components, and detailed specification of the connections between them. These capabilities are described by relating models to potential systems built to meet the specifications, an approach typical in specifying semantics of formal languages. The paper is intended for those concerned with increased precision in SysML concepts, such as builders of model analyzers, checkers, and other automated support for systems engineering. It also enables more reliable interpretation of the SysML models generally, for example, during manufacturing and other stages of the product lifecycle. Examples are given in SysML notation for each concept. © 2013 Wiley Periodicals, Inc. Syst Eng 17: 392–406, 2014

Key words: Model-Based Systems Engineering (MBSE); Systems Modeling Language (SysML); modularization; components

1. INTRODUCTION

One approach to specifying complex systems is to hide internal components, and restrict which portions of each component can be connected to others in the system. Component designs specify which portions are available for interconnection to other components, and system designs specify interconnections that make use of the available portions. It requires additional engineering effort to choose component boundaries, identify externally available portions, and connect those portions with other components, but for large systems the cost is made up in the efficiency of the entire design process. The ways in which components can be connected is reduced, partly by hiding portions of the components, and partly by limiting how externally available portions can be connected to others. This enables engineers to design systems incrementally, with hidden complexity addressed later or separately.

The externally available portions of components and their interconnections can have a complex structure; for example, connectors on computers have pins and other parts, while nuts and bolts have threading surfaces. Interconnections between

* E-mail: conrad.bock@nist.gov

these structures are also usually complex, for example, the way computer connector pins are linked together, or which nuts and bolts will mate properly via their threading surfaces. Sometimes the externally available portions are expected to be used in particular ways, or perhaps not used at all despite being available. For example, connectors on computers sometimes have holes for mechanical attachments, rather than electrical, and some of the protruding metal surfaces are not expected to be connected at all. Similarly, some of the surfaces of a bolt are for wrenches to be applied, while others are for threading onto nuts, and the top of the bolt head is not expected to be used for anything. Externally available portions of components are also intended to interact with other components of the system in specific ways. Items might be expected to flow into or out of the available portions, which could include data, material, or energy. The available portions might expect other components to have particular characteristics, such as shapes that mate, or relative temperatures that enable heat transfer. The externally available portions of components and their interconnections are complex in themselves, and must be managed to design effectively.

This paper describes support for the design techniques above in the Systems Modeling Language, version 1.3 (SysML) [OMG, 2012a]. SysML is an extension of the Unified Modeling Language (UML) for systems engineering [OMG, 2011], issued by the Object Management Group in cooperation with the International Council on Systems Engineering (as OMG SysML™). Many concepts and notations in SysML are shared with UML, but this paper will describe them all as SysML. The primary SysML construct for specifying systems and their components is blocks, which have ports describing externally available portions of the system or component. Interconnections between component blocks and ports are modeled in the internal structure of system and other component blocks. Section 2 describes blocks, including internal structure, and Section 3 covers ports.

This paper enables more reliable interpretation of SyML models by describing the systems and components that could potentially be built according to SysML constructs, an approach typical in specifying the semantics of formal languages [Genesereth and Nilsson, 1987]. It is intended for those concerned with increased precision, such as builders of model analyzers, checkers, and other automated support for systems engineering. These areas do not have common mathematical formalism that could be used to explain SysML, but all of them take the general approach of formal semantics, and benefit from explanations that relate models to instances conforming to those models. For example, in SysML the systems or components that could potentially be built according to the specifications of a block are called the instances of the block. The paper explains blocks by enumerating characteristics their instances can have, following the approach of formal semantics. Due to its focus on relating models to instances, the paper also enables manufacturers and other participants in the product lifecycle to more accurately understand what blocks specify and whether they have designed factories and other systems properly to meet the specifications. This paper differs from other formalizations of SysML [Berardi et al., 2005; Bouabana-Tebibel et al., 2012; Linhares et al., 2007; Graves, 2010] by explaining concepts in simple but well-defined terms accessible to advanced modelers and modeling tool builders concerned with precise understanding of SysML.

2. BLOCKS

Blocks specify components of systems and their assembly into entire systems. Section 2.1 covers the basic meaning of blocks, while Sections 2.2 and 2.3 show how blocks specify characteristics of components and systems and how they are linked together, respectively. These first three subsections introduce terminology for SysML models as well as for the systems built from them, as reflected by the pair of terms in their titles (the first term is about models and the second is about systems built from them, such as "block" and "instance" in Section 2.2). The subsections use the terms of the systems as built to give a precise interpretation of the terms used in modeling. These definitions are the basis for explanations in the rest of the paper. Sections 2.4, 2.5, and 2.6 describe how flows and other interactions between system components are specified. Section 2.7 covers the assembly of components into larger components or systems.

2.1. Blocks and Instances

Blocks specify individual systems that each conform to the block as a specification (*instances* of the blocks). For example, a block for cars might specify that they have four wheels, which John's and Mary's cars do, making their cars instances of that block, or the block might require something not all cars have, such as four-wheel drive, which might mean John's car is not an instance of the block, but Mary's is. Things can be instances of more than one block at the same time, for example, Mary's car might be an instance of a block for cars and a block for four-wheel drive cars. Mary's car conforms to both specifications, because it has four wheels, as specified by both blocks, and has four-wheel drive, as specified by the block for four-wheel drive cars. Figure 1 illustrates this with the SysML notation for blocks at the top (rectangles with "«block»" at the top), and for instances at the bottom (labeled with underlined text). The blocks are shown in a *block definition* diagram, indicated with the "bdd" abbreviation in the upper left of the frame (the rest of the article will omit the frame and «block» indicator for brevity). The dashed arrows indicate

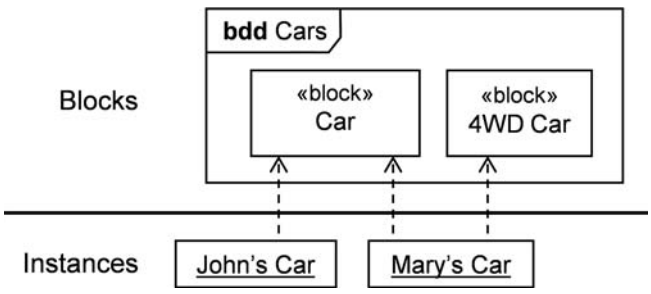

Figure 1. Blocks and instances.

which blocks model which instances (these arrows, solid bar, and labels on the left are not SysML notation).

Things can be instances of some blocks at one time, and others at other times. For example, John might be an instance of a block for students one year, and no longer an instance of that block next year, becoming an instance of a block for engineers instead. Blocks might have no instances at all, either intentionally or not. For example, a block for perpetual motion machines has no instances, and blocks for car designs that turn out to be unbuildable will also have no instances. Blocks only establish conditions for things to be instances; they are not the same as their instances [Bock et al., 2010].

Individual things might be intended to be instances of particular blocks ("specification" blocks), or blocks might be intended to have particular things as instances ("as-built" blocks). For example, cars coming off assembly lines are supposed to be instances of particular specification blocks, and usually the assembly lines will be modified if they are not, while the instances of as-built blocks for cars are supposed to be the cars coming off assembly lines and the blocks are modified if they are not. The difference between specification and as-built blocks is whether the blocks or the instances are changed to maintain their relationship.

Specification blocks usually specify only some aspects of individual things, and can be combined with other blocks into complete specifications. For example, a block limiting car emissions is only concerned with the chemical composition of exhaust, not design of the drive train. Similarly, a car block specifying the drive train probably does not mention emissions. These blocks are models of cars, even though the blocks are incomplete, and individual cars can be instances of them. For example, if John's car meets the limitations of the emissions block, then it is an instance of that block, otherwise not, and similarly for the car block specifying the drive train.

Blocks specifying some aspects of a system can be combined into complete specifications if they are consistent and complementary. One way to combine blocks is by how widely they apply (*generalization*). For example, a block representing emission limitations might apply to more cars than a block for cars having a particular kind of drive train. Figure 2 shows the SysML notation for generalization (arrows with closed, unfilled heads). The Car block generalizes other blocks that specify some aspects of cars, in terms of emissions, materials, and shape (a sans serif font is used in this paper for names of model elements). This means the specifications of Car apply more broadly than the blocks below it, for example, Car might specify the number of wheels, which would apply to all cars, not just the ones with low emissions. The partial specifications in blocks for emissions, materials, and shape are combined by generalizing to a single GoodCar block, which means good cars have the specified emissions, materials, and shape.

The AsBuiltCar block models things that are produced by car factories. As-built cars might not meet any of the specifications of the other blocks, even the one for cars in general, because as-built cars might not be manufactured properly. However, some (and hopefully many) instances of AsBuiltCar might also be instances of the other blocks in Figure 2, having low emissions, recyclable materials, and streamlined shapes. Products of car factories not meeting any of the specifications for the other blocks in Figure 2 would only be instances of AsBuiltCar.

Another way to illustrate blocks and generalization is Venn diagrams, as shown in Figure 3 (this is not SysML notation), reflecting the blocks and generalizations in Figure 2. Block names are shown labeling ellipses that are imagined to enclose instances of those blocks (instances are omitted in the figure for brevity). Ellipses entirely contained in others are for blocks that are related by generalization, where the larger ellipse is the more general block. For example, the ellipse for LowEmissionCar is entirely contained in Car. The Venn diagram highlights that all instances of a block are instances of its more general blocks, because all the instances enclosed by an ellipse are also enclosed by its containing ellipses. For example, all instances LowEmissionCar are instances of Car. Venn diagrams also highlight that generalization does not completely constrain the instances of blocks. For example, GoodCar is generalized by LowEmissionCar, 70%RecyclableCar, and StreamlineCar, but not all cars with low emissions, recyclable materials, and streamlined shapes are instances of GoodCar, because the ellipse for good cars does not take up the entire intersection of the other three ellipses. This reflects some other conditions required for good cars that are not captured in Figure 2. The generalizations of GoodCar only give some of the characteristics of good cars, because generalization only requires that all the instances of a block

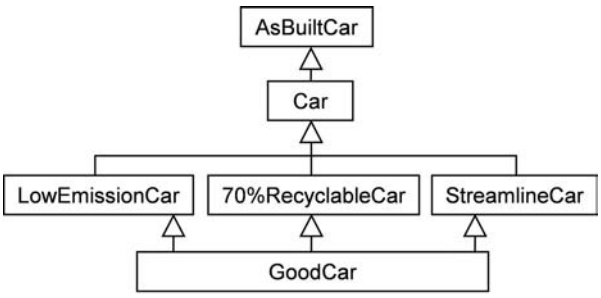

Figure 2. Generalization.

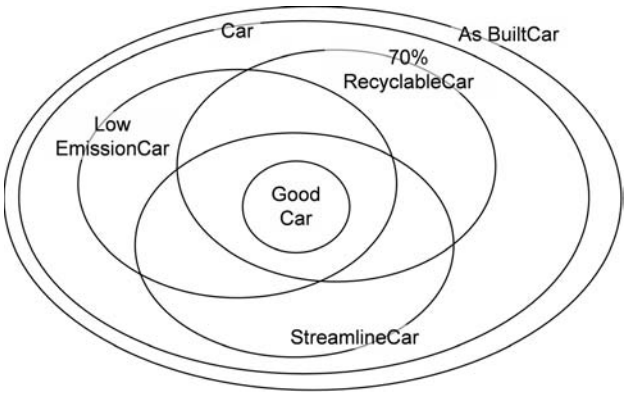

Figure 3. Generalization and Instances.

be instances of its generalizations, not that all the instances of the generalized block be instances of the block.

2.2. Properties and Values

Blocks have several kinds of *features*, some related to how blocks interact with other blocks (*behavioral* features), and others giving characteristics of blocks themselves (*structural* features):

- Behavioral features specify tasks that can be undertaken at the request of other blocks, including services (*operations*), or notifications that can be received from other blocks (*signal receptions*). For example, pumps may have an operation to move water, and a reception notifying them when the target location is full.
- Structural features specify a wide variety of nonbehavioral aspects of blocks, mostly *properties*. These include simple alphanumeric aspects (*value* properties), such as weight, manufacturer name, or color, and complex aspects, such as geometry, meshes; and other blocks, which might be contained (*part* properties), or not (*reference* properties).[1]

Properties have a dual purpose: specifying the kind of things they provide (property *types*), as well as specific things given in each instance (property *values*, not to be confused with value properties). For example, the type of a temperature property might be a number in Centigrade units, while the value for a particular instance will be a particular number, such as –2 ºC for one of the ice cubes in John's freezer. The value might be slightly different for other ice cubes in John's freezer, and might be more different for an ice cube in Mary's freezer, but the value will always be a number in Centigrade units, as required by the property's type. If the value does not conform to the type, for example, if it is given in Fahrenheit, or includes letters, then the value is violating the type and is invalid.

Properties can be typed by blocks, in which case their values are instances of those blocks. For example, the type of a property identifying a car's front left wheel might be a block representing wheels with a particular size range, as shown in the upper part of Figure 4. Properties appear in a list within block rectangles, with the property name to the left of the colon and the type to the right (the type name includes the numeric range and unit for brevity). The value of the property for John's car will be a particular wheel (instance of the wheel block), of a particular size, as shown in the lower part of Figure 4 (instance names at the top of rectangles are underlined, and are used to the right of "=" to show property values). The value of the same property on Mary's car will be a different wheel with a different serial number (another instance of the wheel block), possibly of a different size, but the values for John and Mary's cars will both be wheels, and be within the size range indicated by the property type; otherwise the values are invalid. Properties with no type can have any value and still be valid, but this might just be an incomplete model, rather than intentionally unrestricted.

[1]The information given in properties can change over time; for example, temperature can change, which might change geometry. They are considered structural because they do not specify exactly how they change.

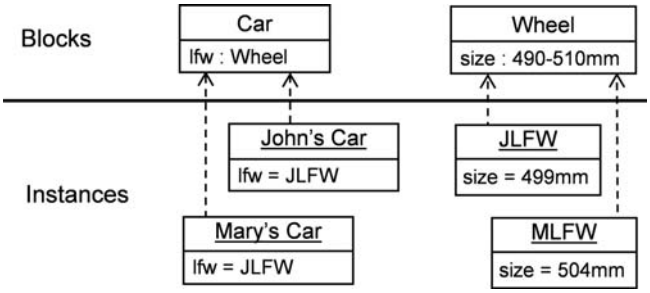

Figure 4. Properties.

Properties can have the same or shared values, even on the same thing. For example, cars have front wheels and powered wheels, and in some cars these are the same set of wheels (front-wheel drive cars), while in others there are more powered wheels than front wheels (four-wheel drive cars). A block representing cars might have these two properties, as shown in the upper part of Figure 5. In some instances, such as John's car, the values of the two properties will be the exact same instances of the wheel block, while other instances, such as Mary's car, more wheels will be powered, as shown in the lower part of Figure 5 (the rear wheels of John's car are omitted for brevity). Properties can be thought of as "roles" of the blocks they are on, such as the front wheels and powered wheels of a car, with their values "playing" the roles, including values that play more than one role. A specialized block for front-wheel drive cars can be defined requiring the values of these two properties to always be the same (see Figure 16 in Section 2.7 about modeling this in SysML).

Sometimes the values of one property are included among the values of another, rather than having exactly the same values (property *subsetting* or property "generalization"). For example, the values of a property for the front wheels of a car will always be included among (be a subset of) the values of a property for all the wheels. Finally, properties can have values that are the same as the things they are on (called "self" properties in this paper). For example, cars might have a property that always has a value that is the same as the

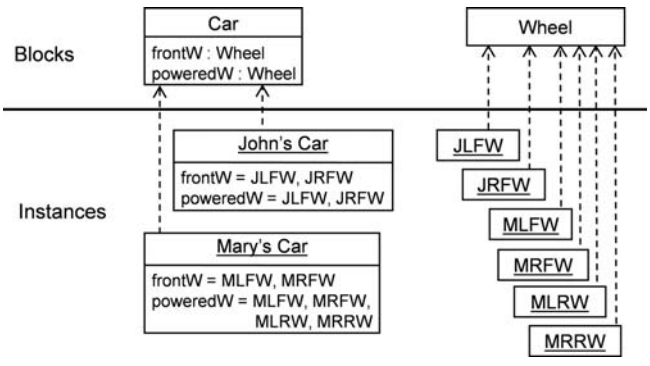

Figure 5. Properties sharing values.

individual car it is on (on John's car it has John's car as a value, on Mary's car it has Mary's car as a value, and so on). This seems redundant, but self properties are important to some approaches to component modeling in SysML (see Section 3.3).

Properties that share values with other properties do not affect the bills of material or "part count" during manufacturing, because each value is only counted once, no matter how many properties have that value. For example, the powered and front wheel properties on John's car as in Figure 5 might each have two values, adding up to four, but if his car has front-wheel drive, the two values are the same. Then the two properties only contribute two instances of the wheel block to the manufacturing bill of material for John's car (the instances needed at the time it is assembled). Modelers might highlight this by choosing one of the properties to be a part property, and the other a reference, but more is needed to ensure they have the same values (see Section 2.7). Self properties also do not affect bills of material, because their values are the thing that is assembled from parts on the bill.

2.3. Associations and Links

Associations specify which properties are about the same relationship between blocks; for example, a property giving the front wheels of a car is about the same relationship as a property giving the car that the wheels are in.[2] Figure 6 illustrates this with the SysML notation for associations (lines between blocks, with a diamond indicating that frontW is a part property). Names of the properties involved (association *ends*) appear near the line ends opposite from the blocks they are on. In this example, the frontW property is the same one as in Figure 5, but shown in a way that makes its related property inCar more obvious. The values of associated properties must be consistent with each other. For example, if a wheel is a value of frontW for a particular car, then that same car must be a value of inCar for the wheel; otherwise the values are invalid.

Associations also specify *links* between instances of blocks (links of associations are analogous to instances of blocks). Property values should always be consistent with links of associations that have the properties as ends. For example, links for the association in Figure 6 between a particular instance of car and its front wheels should be consistent with the values of the frontW property giving the front wheels of that instance of car. Properties typed by blocks can informally be said to "link" instances of blocks, even if they are not ends of associations, because the properties have values on instances of blocks, where the values are instances of other blocks.

Association blocks are both associations and blocks, enabling their links to have the characteristics of instances, including properties. For example, an association block linking cars to their wheels might have a property indicating the rate that energy is flowing to the wheels at any given time (power), as illustrated in Figure 7 with the SysML notation for association blocks (a dashed line with no arrowheads between the association line and block rectangle). This example has energy going to the front wheel of John's car at different rates, perhaps because the car is going around a curve at this particular moment (property values can change over time; see footnote 1 in Section 2.2). The instances at the bottom are links of the association FrontWheelInCar (the dashed arrows between the instances and FrontWheelInCar are omitted for brevity). Association blocks can also have part and reference properties (see Section 2.7).

[2]UML associations can involve more than two properties, but only two are allowed for SysML compliance.

Figure 6. Associations.

2.4. Flow Properties

Flow properties specify things that might flow into or out of a block, as the type of the property. For example, a block representing automobile factories might have a flow property with cars as its type, indicating that cars flow out of factories, and similarly the same block might have incoming flow properties typed by engines or frames, indicating the kinds of things that flow into the factories, as shown in Figure 8. Flow properties are considered structural, even though they have a behavioral aspect, because values of flow properties are the particular things flowing in or out at a particular instant (*items* or flowing instances), rather than over time. For example, the outgoing flow property of Factory in Figure 8 has Car as its type, but for a particular factory at a particular instant, the value will only be the individual cars with particular identification numbers (instances of Car) leaving the factory at that particular time. The same applies to flows of nondiscrete things, like liquids or energy, where instances flowing in or out at a particular time are infinitesimally small "slices" of liquid or packets of energy. These slices or packets conform to the property's type, indicating the kind of liquid or energy flowing, otherwise the values are invalid.

Items flow along links between instances of blocks. For example, factories in Figure 8 might be linked to suppliers via an association between factory and supplier blocks. An out-

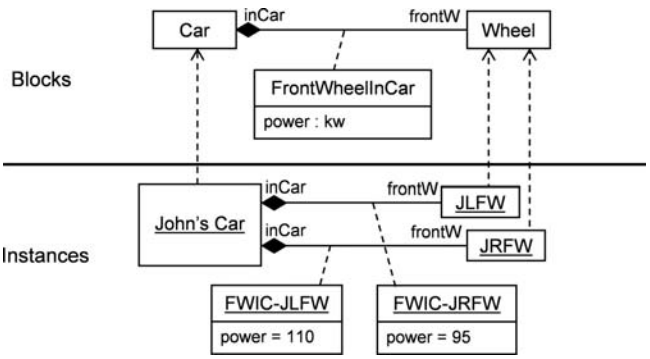

Figure 7. Association blocks.

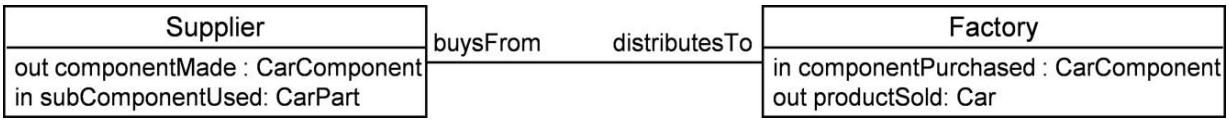

Figure 8. Flow properties.

going flow property for components leaving suppliers can correspond to an incoming flow property for components arriving at factories. When individual components with identification numbers leave a particular supplier, they will go to the particular factories linked to that supplier via the association between suppliers and factories. Similar examples apply to flows of nondiscrete things, like energy or liquids. Outgoing flow property types must be the same or more specific than corresponding incoming flow property types for flow to occur. For example, a factory block with an incoming flow property typed by wheels might be associated with a supplier block with outgoing flow property typed by engines, but the engines will not flow between suppliers and factories linked by this association, because wheels are not engines. See Section 2.5 for restrictions on which links items can flow across. See Section 2.7 for other ways to create links for flows.

2.5. Item Flows

Item flows specify items that flow along associations between blocks. Item flows are *realized* by associations, restricting the flows to occur across links of the realizing associations. This is notated with filled triangles overlaying the association lines and pointing in the direction of flow, as shown in Figure 9. Labels above item flow triangles indicate the kinds of items that might flow. Item flows can be specified without corresponding flow properties. For example, the top of Figure 9 shows an item flow realized by an association between blocks for factories and suppliers that do not have flow properties. A complete model must have flow properties for items to flow, and they must be consistent with each other and with corresponding item flows. Flow properties specify flows without restricting the associations across which flows will occur, while item flows specify flows along particular associations.

The kinds of items flowing can be the same, more specific, or more general than the flow property types on the blocks. For example, in the middle of Figure 9 the kinds of items flowing are engines, small engines, and power sources, which are consistent with the flow properties typed by the blocks for factories and suppliers. Only the middle item flow (SmallEngines) would restrict the possible flowing items more than the types of the flow properties (suppliers could only send small engines to factories), while the top and lower item flows do not restrict the flow more than the types of the flow properties (engines are power sources). The latter kind might appear when the item flows are defined before the flow properties or when the item flows or flow properties are defined on different blocks related by generalization. The kind of item flowing cannot be completely unrelated to the flow property types. For example, in the middle of Figure 9 the kind of item flowing could not be trucks.

More than one association between blocks can realize item flows, with specific kinds of items intended to flow between different instances of the blocks, as shown at the bottom of Figure 9. The blocks for factories and suppliers have two associations between them, one for the flow of new engines, another for the flow of remanufactured engines. The item flow for new engines is realized by the association for new engine distribution, and an item flow for remanufactured engines is realized by the association for remanufactured engine distri-

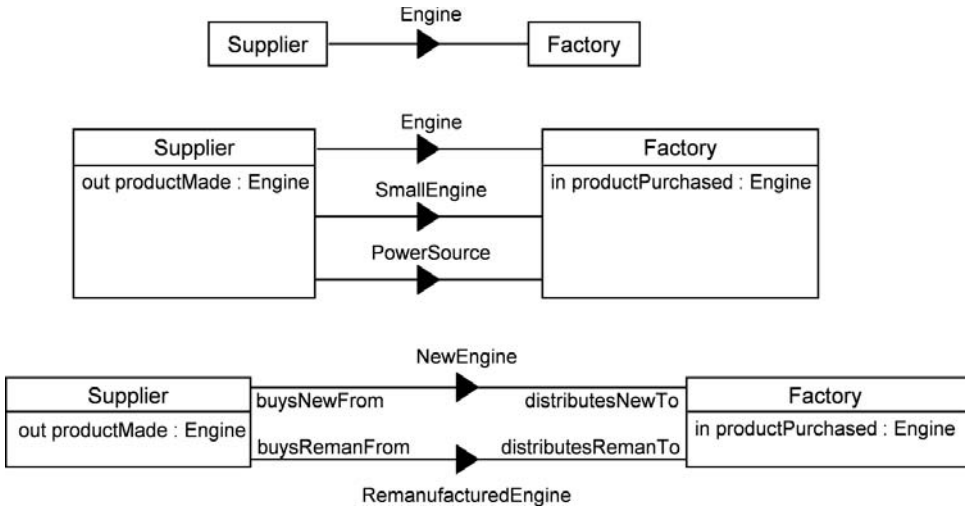

Figure 9. Item flows.

Figure 10. Directed features.

bution. The links of these associations reflect which particular factories are consuming which kind of engines, and from which particular suppliers engines of each kind should come. See Section 2.7 for other ways to realize item flows.

2.6. Directed Features

The features defined on blocks in previous sections are for other blocks to use (*provided features*), while this section introduces features on blocks that are used by those same blocks, but defined on related blocks (*required features*). These are collectively called *directed features*. Figure 10 shows an example of a block for pumps with a required property and operation that things linked to them are expected to support (isFull and isEmpty). These indicate whether those things are full and empty, respectively. Pumps can use the property and operation to determine when to stop or start delivering fluid to things they are linked to, respectively. The required properties and operations on pumps must be matched by provided properties and operations on the linked things, which in this example are tanks (features that do not indicate provided or required are taken as provided).[3] Individual pumps get the values of provided properties on the particular tanks linked to each pump via the association between pumps and tanks, or invoke provided operations on those tanks. Required features do not specify which property or association will link to things supporting the required feature, but at least one must exist for the model to be complete. See Section 2.7 for other ways that links are specified for directed features.

Properties and behavioral features can be both provided and required, which means the features are supported by the block that defines them and by related blocks. For example, in Figure 10 the isInService property is provided and required on pumps and tanks, enabling each to find out if the other is ready to perform its function.

2.7. Internal Structure

Properties and associations are not specific enough to properly model many applications, especially systems of components. For example, an association between blocks for engines and wheels, as shown in the top part of Figure 11, allows links between the engine in one car and wheels in another, as shown in the bottom part of the figure between the engine in John's car and the wheels in Mary's. This is because associations apply to all instances of the blocks they relate. Adding associations between the block for cars and the blocks for engines and wheels does not affect the association between engines and wheels. The three associations are independent of each other, and do not reflect that engines drive wheels only within each individual car, not between cars. More examples of the deficiency of properties and associations in modeling systems are in Bock [2004].

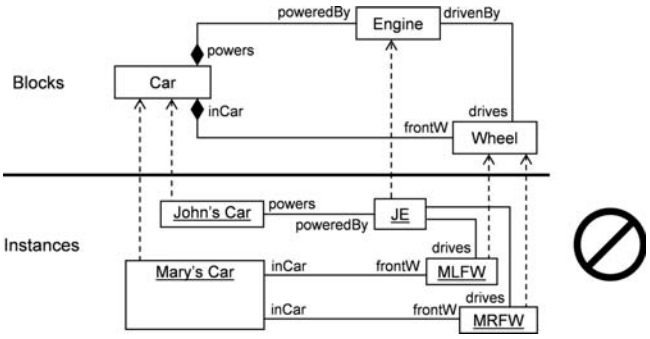

Figure 11. Antiexample for assembling components.

The *internal structure* of blocks addresses these problems by introducing new relations between properties that apply associations in the context of each instance of a block separately (*connectors*). In the example of Figure 11, the properties of a car block identifying the engine and wheel in each car (poweredBy and frontW) can be related by a connector using the association between engines and wheels (drivenBy/drives). This limits links of the association to engines and wheels in the same individual car, not engines and wheels in different cars.[4] Connectors only apply associations to link instances identified by the same "context" instance, in this example, an individual car with a particular identification number.

Internal structure has its own *internal block* diagram, indicated with the "ibd" abbreviation in the upper left of the frame, as shown in the top part of Figure 12. The entire diagram notates a single block Car, as indicated in the upper left of the frame. The rectangles in internal block diagrams are notation for properties of the block, rather than blocks as in block definition diagrams. Property names appear in the rectangle labels to the left of colons, and the property types to the right. These properties can be ends of associations, as shown in Figure 11. The lines in internal block diagrams are notation for connectors, rather than associations as in block definition diagrams. Connectors are adorned with information about their associations, specifically the association end names (defined in Fig. 11 in this example). The effect on instances is illustrated in the lower part of Figure 12. In this example, the drivenBy/drives association is limited to links between engines and wheels in the same car, specifically, the engines in John's and Mary's cars are linked to the wheels in their respective cars, rather than each other's.[5]

Internal structure affects capabilities that depend on links, in particular flow properties, item flows, and directed features

[3]Flow properties are not directed features, but are effectively always provided and required, because related blocks are expected to have similar flow properties in the opposite direction (see Section 2.3).

[4]Associations used by connectors might have links that are not specified by connectors, but these are created separately from internal structures. If links are only created due to connectors, then internal structures effectively restrict how associations are applied.

[5]Figure 12 has instance arrows only for John's and Mary's cars, because the only block shown is for cars; the other rectangles are properties, which do not have instances.

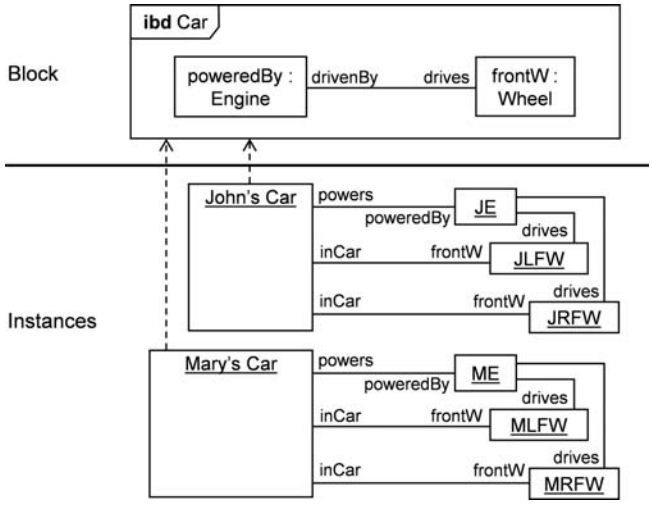

Figure 12. Internal structure.

(see Secs. 2.4, 2.5, and 2.6 about these). They use links to determine which instances items flow between, and which instances might use provided features, or need to support required features. The internal block diagram at the top of Figure 13 shows energy flowing between the engine and front wheels of a car. The item flow is realized by a connector, rather than by an association as in block definition diagrams. This limits the flow to occurring between engines and wheels in the same car, rather than between engines and wheels in different cars, because the connector limits links to each car instance separately. The item flow in Figure 13 is supported by flow properties on the block for engines and wheels, as shown in the block definition diagram at the bottom of the figure.

Similarly to flows, the engine might require wheels to provide information, such as whether they are slipping, via properties or operations, such as the operation on wheels for determining if they are slipping (isSlipping, in the lower right of Fig. 13). The engine accesses this operation on wheels, but only the wheels it is linked to (see Sec. 2.6). In this example, links between engines and wheels are specified by connectors, which limit the links to engines and wheels in the same car.

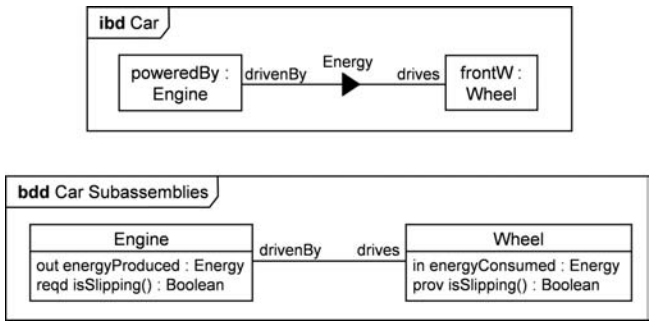

Figure 13. Item flow across connectors.

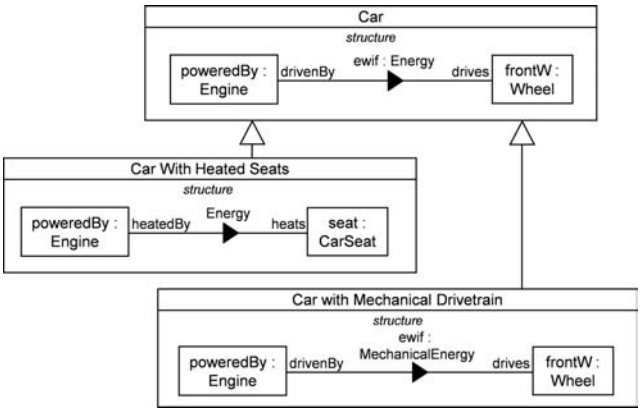

Figure 14. Generalization of internal structure.

This ensures the engine finds out whether the front wheels are slipping, rather than the back wheels, or wheels in other cars.

Internal structure can be combined with generalization to specify connectors more or less broadly as needed (see Sec. 2.1 about generalization). For example, Figure 14 is a block definition diagram showing generalizations between blocks that have internal structure displayed in compartments of the blocks. The general block at the top is the same as the one in the internal block diagram for cars in Figure 13 (except for additional information on the item flow; see below). The specialized block in the middle of Figure 14 has internal structure adding a connector for some of the engine's energy to heat the seats. The connector in the more general block for cars applies to cars with heated seats also, due to the generalization arrow, so the front wheels in cars with heated seats will also receive energy.

The specialized block on the bottom of Figure 14 restricts the flowing energy to be mechanical, which requires an *item property*, notated by a property name to the left of a colon in the item flow label. The type of an item property is the kind of items flowing, shown to the right of the colon. Like all properties, the values of item properties must of the kind specified by the property types, which in this example is energy, or specifically mechanical energy. The values of item properties on instances of blocks are the items actually flowing at any particular time across links specified by connectors realizing item flows. In this example, these values are quanta of energy flowing across links between engines, seats, and wheels in each car separately. The item property in the more general block for cars applies to cars with mechanical drivetrains also, due to the generalization arrow, but the specialized block narrows the type of the item property to indicate mechanical energy is flowing.[6]

Association blocks (see Sec. 2.3) can have internal structure, helping to hide complexity between blocks in the same way internal structure does within blocks. For example, an association for energy delivery between engines and wheels

[6]This is done with *property redefinition* in SysML, which has the same effect as property subsetting (see Sec. 2.2), but enables the property type to be specialized. The redefinition notation is omitted from Figure 14 for brevity.

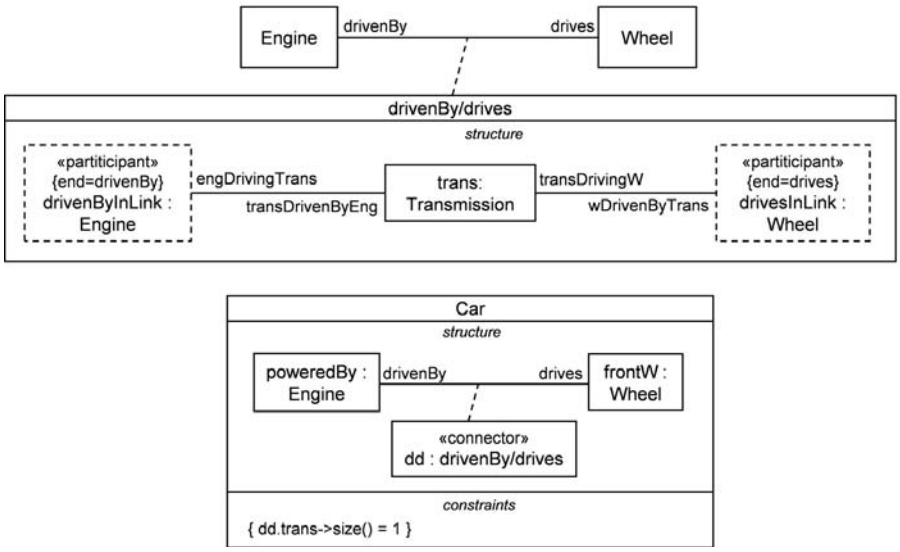

Figure 15. Association blocks with internal structure.

involves many other blocks and associations, representing components such as the transmission, differential, shafts, and the associations between them. Figure 15 is a block definition diagram showing part of this for the association between blocks for engines and wheels from Figure 11. The top of Figure 15 shows a drivenBy/drives association block using the notation from Figure 7 in Section 2.3, but with a compartment for the internal structure of the association. The property rectangles marked with "«participant»" have values that are the instances linked together by instances of the association block (instances of engines and wheels in the example). The expressions in curly brackets identify which association end the participant properties correspond to. Participant property rectangles have dashed borders to identify them as reference properties, because the instances linked by associations are "outside" the association, rather than contained by them (see Sec. 2.2 about reference properties). The third property rectangle is for the transmission, which is "inside" the association. Two connectors ensure the transmission is linked to the engine and wheel at the ends of each instance of the association (the block diagram for associations is omitted for brevity).

The internal structure of a block for cars at the bottom of Figure 15 uses the association block defined at the top. The property rectangle marked with "«connector»" has values that are instances of the drivenBy/drives association block, as indicated by the name to the right of the colon. The expression in curly brackets in the constraint compartment at the bottom ensures the same transmission is used for both front wheels. It is written in the Object Constraint Language (OCL) [OMG, 2012b] and uses the name of the connector property (dd) to refer to the instances of the association block, as well as the name of the property identifying the transmission (trans). The dot operator gives a list of the transmissions in the instances of the association block, and the size operator gives the length of that list, which is constrained to be exactly 1. This limits the transmissions in each car to be exactly 1, rather than a separate transmission for each front wheel.

Item flows could be realized by the associations and connectors in Figure 15 as they are in Figures 9 and 13. For example, the association between blocks for engines and wheels at the top of Figure 15 could realize an item flow for energy, as could the connectors inside its association block between the engine, transmission, and wheels. Modelers should define these consistently with each other, but SysML does not specify a methodology they must follow. For example, if the associations had properties for the rate of flow of energy, as Figure 7 does (power), and the association block constrained the values of this property to be within a particular range, such as 0–100 kW, then modelers should check that the ranges for these properties on the associations between engines, transmissions, and wheels are the same or narrower than the range, such as 0–90 kW, rather than wider, such as 0–110 kW. This reflects that the power range on the association between engines and wheels is an overall restriction that the flows in the association block must adhere to.[7]

SysML provides a special kind of connector for specifying that particular properties have the same values in each instance of a block (*binding* connector) (see Sec. 2.2 about properties sharing values). In the example of Figure 5, Section 2.2, the block representing cars in general has a property for front wheels and a property for powered wheels, but in front-wheel drive cars the front wheels and powered wheels are the same. An internal block diagram for front-wheel drive cars can model this with a binding connector between these two properties, as shown in Figure 16 by the «equal» keyword

[7]Using item flows on associations and the connectors inside them is informally called "item flow decomposition," but item flows cannot be decomposed because they are not blocks, and the relationship between the kinds of things flowing along associations and those flowing along connectors in the associations is not necessarily decomposition; see other examples in Section 3.

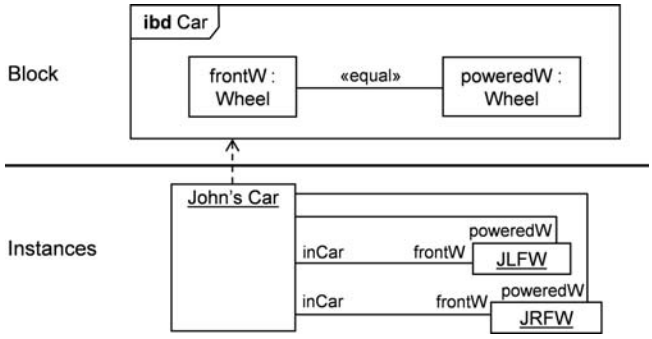

Figure 16. Binding connectors.

on the connector. This indicates values of the front wheel property in each front-wheel drive car are the same as values of the powered wheel property in that same car (the same instances of the block for wheels), rather than other cars. The effect on instances is shown in the lower part of Figure 16. The same two wheel instances (JLFW and JRFW) are linked as front wheels and powered wheels to John's car.[8] Binding connectors highlight properties that do not affect manufacturing bills of material, and this can be further emphasized using reference and part properties together (see Sec. 2.2 about reference and part properties). For example, in Figure 16, the property rectangle for front wheels or powered wheels could have been dashed to emphasize that it does not contribute to bills of material. The values of bound properties must be instances of both property types, either directly or by generalization (see Section 2.1 about generalization).

3. PORTS

Ports specify externally available portions of system components, a critical capability for achieving the benefits of componentization (see Sec. 1). This section describes ports in terms of the concepts from Section 2. Section 3.1 covers the basic meaning of ports and how they contribute to assembling components. Section 3.2 describes how complexity of ports themselves can be managed. Section 3.3 shows how ports appear from inside the blocks that have them.

3.1. Port Properties

Ports are a special kind of property, and like all properties, can be typed by blocks, but unlike other properties, the features and associations of these blocks are explicitly available to other blocks (some ports can also have internal structure; see Sec. 3.3). The features can be any of the kinds described in Section 2, with limitations in some cases, and the associations can have internal structure. Figure 17 shows ports using SysML notation, at the top in a block definition diagram (small rectangles on the boundary of block rectangles labeled in the same way as properties), and at the bottom in an internal structure diagram (small rectangles on the boundary of property rectangles). The connector in the internal block diagram is for the association in the block definition diagram (the association information on the connector is omitted for brevity). Ports are linked by connectors rather than associations, because ports are properties rather than blocks.

Features of ports are externally available via links between the values of port properties and instances of other blocks, which might be values of other port properties (see Sec. 3.3 about the values of port properties). The links can be specified by connectors in internal structures between ports and other properties, including other ports, as in Figure 17. In this

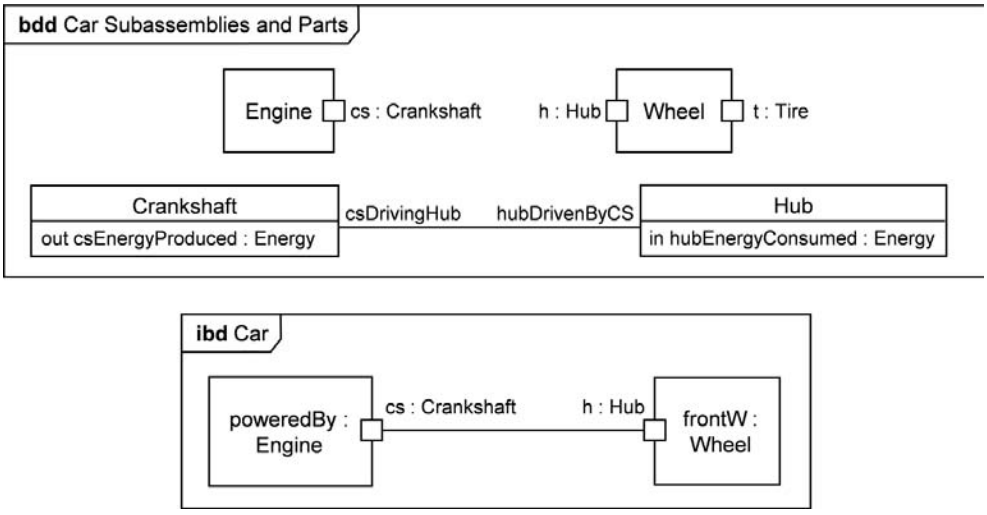

Figure 17. Ports.

[8]Binding connectors do not identify an association to apply, but are equivalent to applying an association that only links instances to themselves, and no others

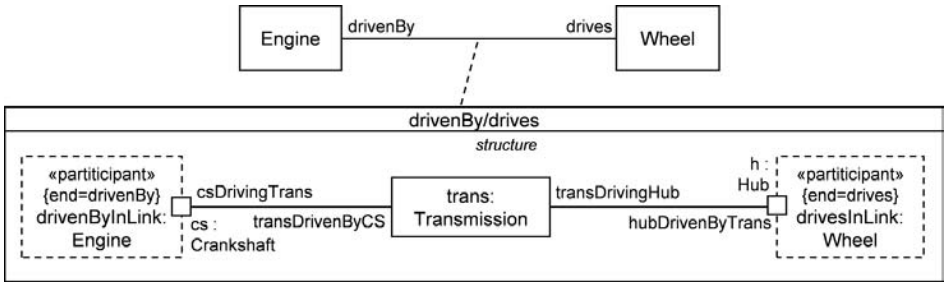

Figure 18. Association decomposition with ports.

example, the flow of energy from the engine is available to the hub, due to the connector in the internal structure diagram. Ports hide complexity in blocks when connectors from outside link only to ports, rather than to top-level properties of blocks. This limits available features to those defined on port types, rather than features of the block having the ports. In addition, ports do not always need to be connected. For example, in Figure 17, the internal block diagram does not have a connector for the tire (it is available only to things outside car). Blocks specializing the block for cars might add connectors (in this example to things outside the car as part of a "total" system specification), but cars built to the internal block diagram in Figure 17 will not have links to the tires.

Connectors can use association blocks that have connectors between ports (see Sec. 2.7 about association blocks having internal structure). For example, the association block of Figure 15 could be modified to connect ports, as in the block definition diagram of Figure 18. The same effect can be achieved by decomposing the connectors of the association block in Figure 15 with their own internal structure, as in Figure 19. The incremental decomposition of Figure 19 enables the model in Figure 15 to hide the complexity of how engines, transmissions, and wheels are related, whereas the approach of Figure 18 puts it all in one diagram. Figure 19 also enables high-level specifications or requirements to be placed on the relationships of engines, transmissions, and wheels, with design refinements meeting those specifications or requirements captured in other diagrams. Combined with generalization of association blocks, alternative design refinements can be captured without modifying high-level specifications or requirements in Figure 15 [Bock et al., 2010]. Figure 18 is simpler than Figure 19 but adds a significant inefficiency to the design process.

3.2. Nested Ports

Blocks typing ports can have features that are other ports (informally called "nested ports"). This hides complexity when specifying externally available portions of systems. Figure 20 illustrates this with a block definition diagram specifying engines mounted on car frames. The blocks for engines and frames at the top of the figure have ports to identify the portions used for mounting. The types of these ports are blocks that have ports for holes in the mounting portions of engines and frames, defined in the middle of the figure using the notational option of showing port names inside port rectangles (decomposition of the association between the blocks for engines and frames is omitted for brevity). The middle of the figure also shows the efm/fem association block between the mounting portions of engines and frames, which has an internal structure connecting the hole ports. These connections are for the fixed association

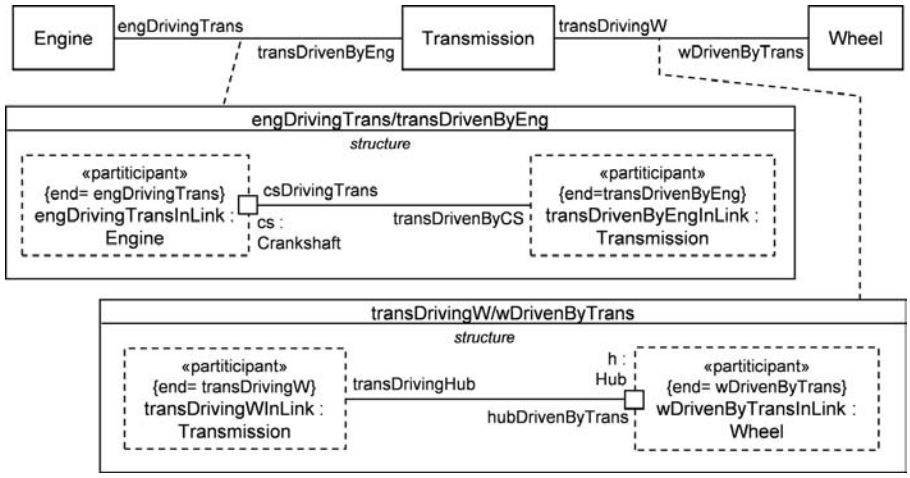

Figure 19. Incremental association decomposition with ports.

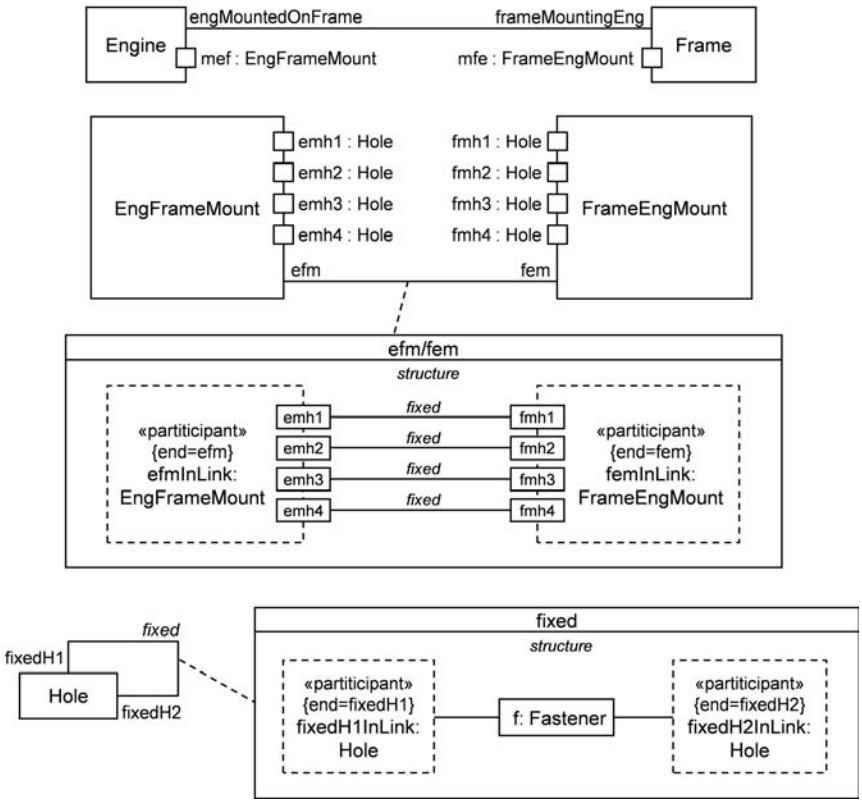

Figure 20. Nested ports.

defined between holes at the bottom left of the figure, which has a further decomposition to specify the fastener on the bottom right. This enables the fastening through each pair of holes to be specified once and reused on all four pairs.

3.3. Full Ports and Proxy Ports

Some ports introduce new elements in systems when they are built (*full ports*), while others only "stand in" for existing elements (*proxy ports*). The values of full ports (as properties; see Sec. 3.1) contribute to manufacturing bills of material, while the values of proxy ports do not. This is because the values of full ports are not shared with other properties (and are not the things the ports are on either; see behavior ports below, and Sec. 2.2 about properties sharing values), while the values of proxy ports are shared with internal part properties or are the same as the things they are on. Although port types determine features available to connectors, port values are responsible for supporting the features (see Sec. 3.1) and reflect whether the port is full or proxy.

This section describes modeling approaches around the two kinds of ports, but SysML does not require a particular approach for using ports, and the techniques described here are only examples for illustration. Modelers can choose between full and proxy ports at any stage of model development, or not at all if the distinction is not needed. Modelers can also make the choice at any level in a block taxonomy, whether on ports of the most general blocks, the most specialized, or in between. The choice is only restricted by constraints on the way the two kinds of ports are related to the surrounding model, as explained below.

Full and proxy ports only need to be distinguished when defining blocks. When using ports on existing blocks, only the features of port types are important. In particular, connectors to ports from outside only "see" port features, not how ports are connected internally. Whether port values are shared with other parts of existing blocks does not affect the port features available outside these blocks, and it would be a very unmodular design for outside blocks to depend on the presence or absence of value sharing in the existing blocks they use. Modelers can switch between full and proxy ports, or not make the choice, at any time, without any effect on other blocks. Tools can omit the notation distinguishing the kinds of port to simplify diagrams when existing blocks are used.

The two kinds of ports capture an important difference in modeling methodology, but do not affect as-built systems. For example, jacks for attachment of computer peripherals are separate from other parts of computers, such as screens and processing units, but ports representing jacks might share their values with other part properties of a block representing computers (proxy), or not (full). Modelers might choose either kind of port to represent jacks, or not make the choice at all, but the computers built to those specifications will have jacks separate from other parts in any case. Figure 21 shows two packages of blocks specifying computers and some of their jacks, using the SysML notation for packages (rectangle

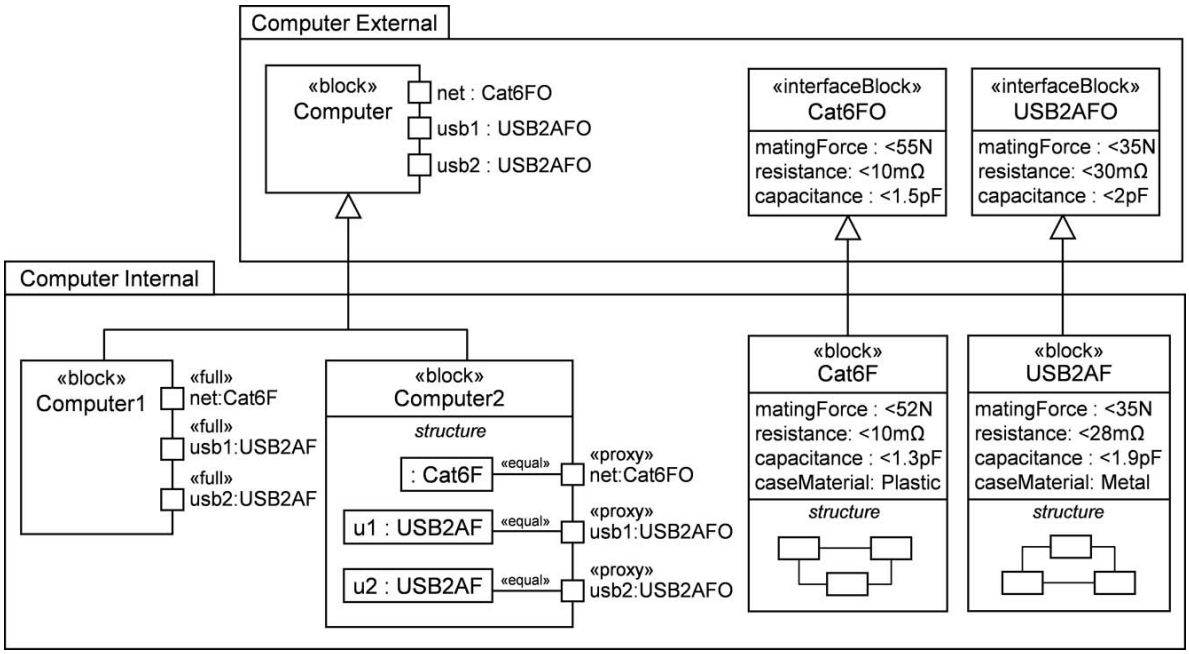

Figure 21. Full ports and proxy ports.

with a tab labeled with the package name). The package on the top is for use by modelers who are not supposed to receive any proprietary information about the internals of the computer or the jacks, such as system integrators. The package on the bottom defines the internals of the computers and their jacks, as might be seen by design or manufacturing engineers. The top package can be sent to modelers or engineers who are not supposed to see system internals, and both packages can be sent to those who are.

The general block for computers in the top package has ports representing jacks, but does not indicate whether they are full or proxy, because the distinction is only of concern when specifying the internals of blocks. The general ports are typed by special kinds of blocks on the upper right (*interface blocks*) that only specify interaction with or between other things, such as the force needed to plug into the jack or the resistance introduced electrical contacts, but not the internals of those other blocks or themselves (including internal behaviors or part properties other than ports). Interface blocks can still have instances with elements corresponding to internal parts, connectors, and behavior, but these must also be specified on blocks that are not interfaces, see below.

Specialized blocks for computers on the lower left in Figure 21 are in the package that defines system internals. The Computer1 block makes the ports from Computer into full ports, while Computer2 makes them into proxy ports[9]:

- The full ports restrict their types to specializations of the general port types, shown on the lower right. The specialized port types are blocks, rather than interface blocks, because they add features of concern only to the jacks themselves, such as casing material and internal structure, rather than features affecting interaction with other things.
- The proxy ports are typed the same as the general ports,[10] but have binding connectors to part properties typed by the specialized port types, indicating their values are the same as the part properties (see Figure 16 in Section 2.7). Interactions with a computer via connectors to proxy ports are the same as interactions with the internal parts they are bound to.

In order for the full ports in Figure 21 to have values conforming to the type of the general ports, and for the proxy ports to have the same values as the internal part properties they are bound to, the interface blocks must have instances with internal elements corresponding to internal parts, connectors, and behavior in the model (see Sec. 2.2 about property values conforming to property types). These internal elements cannot be specified on interface blocks, but can be specified on the specializations of interface blocks, as shown on the lower right of Figure 21. The specialized blocks are used as the types of full ports, and as the types of internal parts bound to proxy ports. Generalization ensures that instances of the specialized blocks with internal specifications are also instances of the interface blocks (see Sec. 2.1 about generalization). Also see the related example in Figure 22.

Computers built according to the two specialized blocks will be the same, because proxy ports do not introduce anything new into the bills of material compared to using full ports. Full ports are simpler to model, but proxy ports are more flexible, because they can be bound to potentially deep nested part properties that participate in complicated internal

[9]This is done with property redefinition in SysML, which has the same effect as property subsetting (see Sec. 2.2), but enables stereotypes to be applied. The redefinition notation is omitted from Figure 21 for brevity.

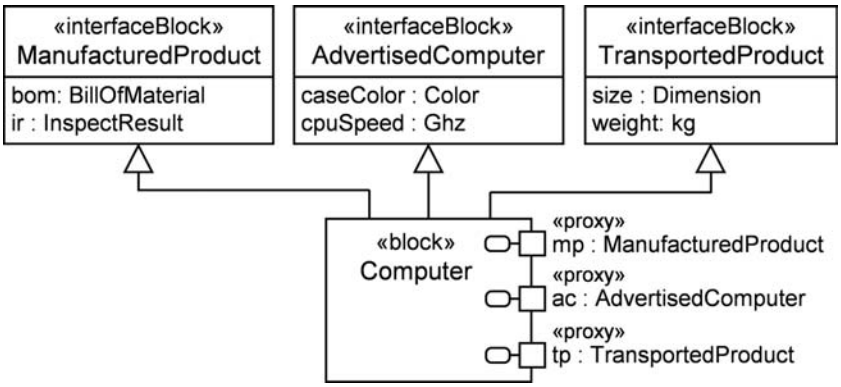

Figure 22. Behavior ports.

structures. In these cases it is graphically easier to define proxy ports separately from the internal parts and bind them together, rather than try to move a deeply nested property rectangle to the boundary of the containing block rectangle. When it is graphically feasible to move an internal part property rectangle to the boundary and define a full port, proxy ports can still be defined as nested ports on full ports to restrict the features available to external connectors.

The values of proxy ports can be the same as the things they are on (*behavior ports*, a kind of self property; see Sec. 2.2), rather than being the same as the values of internal part properties. When features of behavior ports are used, it is as if the same features were used on the instance having the ports. For example, computers might have features for various purposes, such as bills of material and inspection results for manufacturing, color and CPU speed for advertising, and size and weight for transportation planning. These features can be separated into interface blocks that generalize the block for computers, and the interface blocks can type proxy ports that are also behavior ports, as shown in Figure 22 using the notation for behavior ports (a line to a small rounded rectangle). The generalizations ensure externally available features specified by the behavior port types are supported on the block for computers. They also ensure values of the behavior ports, which are the instances of the block for computers, will conform to the type of the behavior ports (see Sec. 2.1 about generalization, Sec. 2.2 about property values conforming to property types, and the related example in Fig. 21). Because these are behavior ports (self properties), things linked via connectors to the port for manufactured products (mp) have bills of material and inspection results available to them, while things linked via connectors to the port for transported products (tp) have size and dimension available.

4. SUMMARY

Blocks and ports are the primary SysML concepts for reducing the complexity of specifying systems through componentization. This paper introduces these concepts and covers new capabilities in SysML that enable them to model a wider range of systems. Blocks specify features of systems, including features of their components and how the components are related to each other via associations and connectors (internal structure). Components can also be related to each other via flows between them, including over associations and connectors, specified with features such as flow properties, item flows, and directed features. Ports are special features of blocks for specifying externally available portions of system components. Ports can also have nested ports when these externally available portions become complex. Ports can be specified as introducing something new into the system (full ports) or as standing in for other parts of the system (proxy ports). The paper explains these constructs by specifying the systems and components that could potentially be built according to them, an approach typical in specifying semantics of formal languages, enabling more reliable interpretation of SyML models. Combined with other capabilities of SysML, such as the internal structure of associations, blocks and ports provide modelers with a powerful and scalable way to specify complex systems.

[10]Proxy ports must always be typed by interface blocks to highlight that they only stand in for other model elements that specify behaviors and nested parts.

ACKNOWLEDGMENTS

The author thanks the members of the Object Management Group SysML Revision Task Force for many discussions on the topics of this paper, as well as Jae Hyun Lee and Donald Libes for helpful comments.

Commercial equipment and materials might be identified to adequately specify certain procedures. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

REFERENCES

D. Berardi, D. Calvanese, and G. De Giacomo, Reasoning on UML class diagrams, Artificial Intell 168(1–2) (October 2005), 70–118.

C. Bock, UML 2 composition model, J Object Technol 3(10) (2004), 47–73.

C. Bock, X. Zha, H. Suh, and J. Lee, Ontological product modeling for collaborative design, Adv Eng Informatics 24(4) (2010), 510–524.

T. Bouabana-Tebibel, S. Rubin, and M. Bennama, Formal modeling with SysML, Proc IEEE 13th Int Conf Inform Reuse Integration, 2012, pp. 340–347.

M. Genesereth, and N. Nilsson, Logical foundations of artificial intelligence, Morgan Kaufman, Burlington, MA, 1987.

H. Graves, Logic for modeling product structure, Proc 23rd Int Workshop Description Logics, 2010, pp. 486–496.

M. Linhares, R. de Oliveira, J. Farines, and F. Vernadat, Introducing the modeling and verification process in SysML, Proc IEEE Conf Emerging Technol Factory Automation, 2007, pp. 344–351.

Object Management Group (OMG), OMG Systems Modeling Language, version 1.3, http://www.omg.org/spec/SysML/1.3, Needham, MA, June 2012.

Object Management Group (OMG), OMG Unified Modeling Language, superstructure, version 2.4.1, http://www.omg.org/spec/UML/2.4.1, Needham, MA, August 2011.

Object Management Group (OMG), Object Constraint Language 2.3.1, http://www.omg.org/spec/OCL/2.3.1, Needham, MA, January 2012.

Conrad Bock is a Computer Scientist at the U.S. National Institute of Standards and Technology, where he is project leader for systems engineering modeling standards. He is co-chair of the Object Management Group (OMG) Revision Task Force for the Systems Modeling Language (SysML). He led work on SysML 1.3's upgrade for modularity, developed SysML's process modeling extensions, and contributes to several other areas of the specification. He is one of the primary contributors to the Unified Modeling Language at OMG, on which SysML is based. He studied at Stanford, receiving a B.S. in Physics and an M.S. in Computer Science.