Structural Decomposition Pattern

Synopsis

Decomposition refers to the process by which a complex problem or system is broken down into parts that are easier to conceive, analyze, develop, or maintain. The word *decomposition* also refers to the product that is the output of the decomposition process. A *str uctural* decomposition is a hierarchical tree structure of things of the same kind (physical, logical or conceptual) representing whole-part relationships. Typically, a whole-part relationship means that certain operations that apply to the whole (e.g., movement in space) may also apply to each of its parts, and that a property of the whole (such as mass) can be inferred from the properties of its parts. Examples of structural decomposition include Product Breakdown Structure (PBS for hardware or software) and Bill of Materials (BOM).	

Pattern Overview

Pattern Status	Tool Version
In Work	SECAE MagicDraw Packages versions 1702SP3-02 or later

Line Organization Owner	Submitter	Point of Contact
3101	IMCE Pattern Consolidation Working Group	Dan Dvorak

Related Patterns
• Characterization Pattern: How to describe the properties of elements in a decomposition • Structural Context Pattern: Within a decomposition, what can be asserted and analyzed • Reconciliation/Abstraction Pattern: How to reconcile different, but valid, representations of information

Skip directly to the SysML examples...

Table of Contents:

Applicability

To help users assess the applicability of this pattern to their work (i.e., to the problem they want to solve or their area of interest), we describe the way in which this pattern addresses a few kinds of common concerns. In particular, we address:

- Content concerns: the kind of content users can capture in this pattern
- Artifact concerns: the kinds of artifacts (documents and views) that can come from this pattern
- Reasoning concerns: the kind of reasoning (analysis) that this pattern is meant to support
- Assumptions: what we expect to be true about the user's situation that is relevant to whether they can or should use the pattern.

Content Concerns

Structural decomposition involves the use of whole/part relationships to define the complete and disjoint set of children which "are a part of" a parent in order to break the system down in to smaller and more manageable pieces.

Clear and concise discussion of a structural decomposition pattern requires that all stakeholders must have a common understanding of the definition and scope of the term. Definitions of the word "decomposition" vary depending on the context (we have no interest in discussing biological decomposition, for example) - but all definitions have at their core the concept of separating or reducing; the transformation from some kind of union into disjoint subsets.

The precise meaning of "structural decomposition" is even slightly problematic in our own domain. For example, the term "structural decomposition" does not appear in either the NASA Systems Engineering Handbook or JPL Systems Engineering Practices documents, and broader searches of the term lead to many informal definitions. While a limited investigation of these yields the common theme of system decomposition as an exercise of dividing a system or problem into smaller systems or problems which are more manageable to describe and analyze, there does not seem to be a universally accepted definition or assertion of formal rules for decomposition of "systems".

To describe a pattern called "Structural Decomposition", then, we will not attempt to provide some unifying definition, but instead to address a few common concerns that we often address using the label of structural decomposition.

What do we include in structural decomposition?

The successive decomposition of a tangible deliverable into its major constituents, their constituents, and so on--what Systems Engineers call a Product Breakdown Structure--is the canonical example of structural decomposition. This structure is distinguished by two properties: first, that every element in the structure is in some sense a product with its own breakdown, and second, that the relationship joining every parent to its children is "is made of". A Product Breakdown Structure is a tree, and each of its subtrees is also a Product Breakdown Structure. This recursive invariance is a hallmark of structural decomposition.

There are other similar tree structures in systems engineering; the goal of this pattern is not to model every type of tree structure, only those comprised of the "is made of relationship. That excludes many types of trees, such as the Work Breakdown Structure. More discussion is included in the advanced questions section.

Very broadly, we will say that structural decomposition involves the use of whole/part relationships to define the complete and disjoint set of children which "are a part of" a parent. We will focus particularly on two important decompositions that we often do in systems engineering:

- **Assembly:** a tree structure resulting from composing leaf elements into physically joined assemblies recursively upward until we have one structure containing all of the leaf elements.
- **Functional Performer:** a tree structure of logical elements, each of which exist to perform functions. Note that in this tree structure it is the logical elements that are being decomposed, not the functions they perform. Thus, this is *not* a "functional decomposition"; the concept of "functional decomposition" or "functional elaboration" will be addressed in another pattern.

We choose to scope the structural decomposition pattern to these two examples of structural decomposition - we do not claim that this is the whole set of structural decompositions.

Decomposition can also be a meaningful concept for non-physical objects. In computer networks, for example, it is common for protocols at different levels of abstraction (e.g., physical, link, network, session, etc.) to be related by encapsulation, which is a form of decomposition.

A Layer 2 Ethernet frame, for example, is a link-layer protocol that is defined to consist of an Ethernet header, a message payload, and an Ethernet trailer. The payload, in turn, might be an Internet Protocol (IP) network-layer datagram, which is defined to consist of an IP header and a payload. This payload might be a Transmission Control Protocol (TCP) session-layer datagram, which is defined to consist of a TCP header and a payload. Each layer composes a datagram by encapsulating a datagram from a higher layer in the protocol stack. Structural decomposition is an appropriate modeling pattern for this situation.

> ⚠ **Note about "Functional Decomposition"**
>
> System engineers sometimes use the term "functional decomposition" to to refer to a tree of functions in which child functions contribute to parent function(s). However, this term should be discouraged because functions don't arise purely by aggregation. One cannot deduce the function of a composition of logical elements from the composition alone; it is also necessary to know how the logical elements collaborate with one another and the manner in which these interactions are exercised. Different collaborations of the same components will perform differently. Since composition/decomposition alone is mute about such distinctions, the term "functional decomposition" is misleading. Something *other* than mere composition is needed for such descriptions.

In summary...

We choose an **"assembly"** structure and a **"functional performer"** structure because these are common ways of decomposing systems into more manageable parts. They represent two very different but equally valid structures which are present in systems engineering work and can be tricky if conflated.

In the assembly decomposition, we can address concerns about the organization, attributes, and process of transforming a flat set of components into a realized solution to a problem. We have a structure in which we can analyze, assess, verify, and test the realization of our design. In the functional performer space, we can assert and assess our intent for the system, grouped by what we want it to do.

This pattern will describe the construction of these two decompositions and provide guidelines and suggestions for managing structural decomposition in general.

Artifact Concerns

When people think of artifacts of a structural decomposition, they often imagine a picture of a tree structure, or sections of a document. While these are indeed things that can be derived from a structural decomposition, this pattern also supports the tree traversals necessary to create other kinds of artifacts:

- Master Equipment Lists or Mass Equipment Lists (MELs)
- Power Equipment Lists (PELs)
- Manufacturing Bill of Materials (MBOMs)

Generic Reasoning Questions

Because the Structural Decomposition Pattern is fundamentally about whole/part relationships, reasoning about structural decompositions is about what things are "part of" other things. You can ask about direct children or transitive children. Conversely, you can ask if something is a direct or transitive parent.

These traversals allow many kinds of part/whole analysis - for example, the artifacts listed in the previous section are analyses which depend upon traversing the tree structure, sometimes doing operations at each node. For example, one must have knowledge of the child elements in order to calculate the mass of the parent. Similarly, one can create a flat bill of materials for a parent by identifying all leaf elements which are transitive children of that parent. In the assembly case, the whole/part relationship explicitly means that you cannot complete the assembly of a whole until you have all of the parts, allowing calculations about assembly schedules and lead times.

If a reasoning question involves "is this a part of that?" then it is relevant to structural decomposition.

Assumptions

An appropriate authority must define the specific meaning of "is a part of" and modelers doing structural decomposition must use that definition consistently across all uses in the scope of the work. We have provided two examples, but the entire set of subtle interpretations of the whole/part relationship is not part of this pattern (for example, see Six Different Kinds of Composition - Odell). We address the representation of whole part relationships and we assume that implementers of this pattern will define exactly what whole/part means for the context in which they assert it to be true.

Pattern Implementation

Here we describe the elements that make up the pattern, their relationships, responsibilities, and collaborations. The solution does not describe a particular concrete design or implementation. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements solves it. The solution is presented in modeling language independent terms and in its SysML embedding.

Generic/Ontology Implementation

This pattern makes use of the abstract concepts of Container and ContainedElement (these have been described in the Structural Context Pattern). Container and contained element apply to a very large set of "containment" relationships; here, we describe a special application of that concept for the purpose of structural decomposition.

When using containment to represent structural decomposition, we do require that at each level of the tree, the tree is constructed by using the same type of relationship and the parents and children have to be the same type (either assembly or logical performer). This can also be described as the relationship being homeomeric ("whether or not the parts are the same kind of thing as the whole", Six Different Kinds of Composition - Odell). Ontologically, we do not provide means for determining whether a structure is homeomeric - that is left up to the implementer.

We have now mentioned that these trees exist within a context or within a view. Ontologically, we say that a structural decomposition exists within a context and is defined by homeomeric part/whole relationships whose formal meaning is determined by the context in which they exist. We leave the definition of what exactly that context is up to the implementer. This is in contrast to the use of "context" in the Structural Context Pattern, which is concerned with the internal context provided by elements in a decomposition.

SysML Implementation

We will first describe the embedding of the ontological concepts into SysML, and then present some examples to ground our concepts.

Embedding

The embedding of Containers and ContainedElements is described in the Structural Context Pattern.

The only new idea we introduce is that these decomposition trees are confined to some kind of "context" or "view". This is a utilitarian decision made to allow the meaning of the contains (part/whole) relationship used in the decomposition to be determined by the particular view. For example, we do not provide a way to tell the difference between a containment relationship that implies physical assembly and one that implies functional performer decomposition. In order to tell, we suggest that the modeler look to the view which contains the tree.

As for what form in SysML that "view" takes, it is still an open question. Our initial recommendation is to utilize namespace containment to keep trees separate (i.e., a model package structure), and whether to simply define for the project the meaning of each decomposition or to create special stereotypes is up to the modeler.

SysML Examples

Pedagogical Example Illustrating SysML Embedding

You can see a pedagogical example of containment in the Structural Context Pattern.

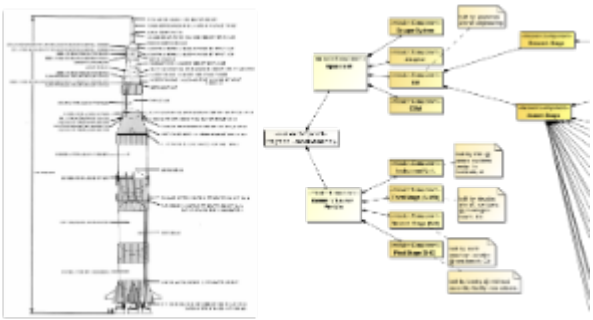
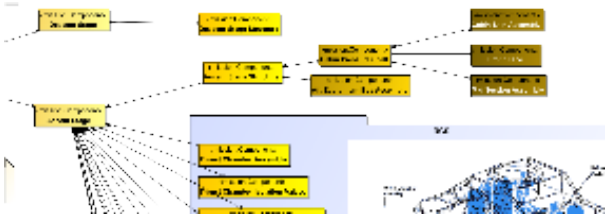
SysML Example: Assembly and Functional Performer decompositions in the Apollo 11 Mission.

The Apollo 11 mission provides good examples of both complex assembly structure and a complex functional performer decomposition and it is utilized here as a good real-world example. Both the launch vehicle and the two spacecraft contained multiple stages and underwent many configuration changes.

Expand to continue reading...

This example shows a partial decomposition of the integrated Apollo 11 flight elements, first in the assembly space and then in functional performer space.

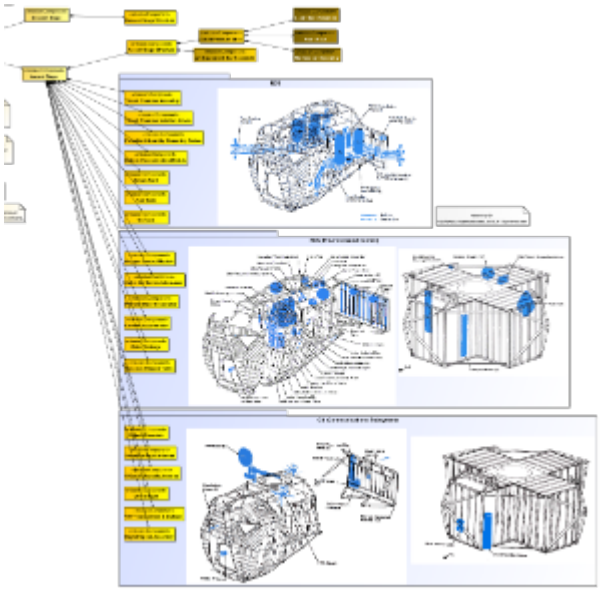
Assembly Space

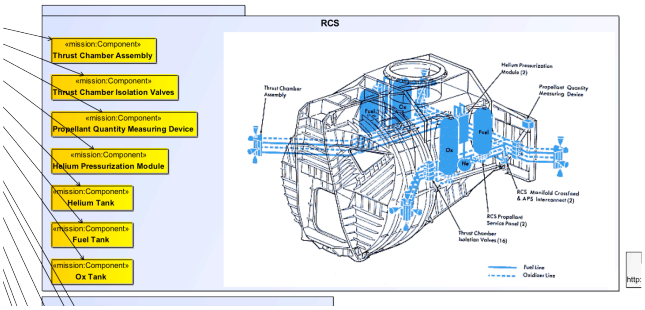
Here we show the decomposition of the integrated Apollo 11 launch stack into its component parts. This assembly structure was derived from historical NASA documents about the assembly processes of both the Saturn V launch vehicle and the Lunar Module. The Saturn V was built in four separate pieces, two of which were so large that they had to be transported from their places of manufacture (California and Louisiana) to Florida by barge. The four pieces (the First Stage, Second Stage, Third Stage, and Instrument Unit) are shown in the diagram to the right. The Saturn V was assembled what was originally called the Vertical Assembly Building at KSC, ideally in order from the bottom stage to the instrument unit sitting atop the third stage. However, if pieces did not arrive in the correct order, "blocks" of the same mass and shape were used to allow the components present to undergo "integration" until the flight component arrived.	
The Apollo 11 "Spacecraft" assembly consisted of the mated Command Module (CM) and Service Module (SM), known as the Command/Service Module (CSM). Although the CSM was considered a spacecraft on its own, the "spacecraft" that was mated with the Saturn V also consisted of the following: an escape system to pull the crewed command module away from the rest of the stack in case of an abort; the Lunar Module (LM) (also a spacecraft in its own right); and the Spacecraft-Launch Vehicle Adapter (SLA).	

To continue our assembly decomposition, we looked into the process of assembly of the Lunar Module. The LM was built by Grumman Aircraft Engineering in New York, and was assembled from a mix of parts manufactured on site and procured from subcontractors. The LM consisted of two assemblies: an Ascent Stage and a Descent Stage. We continue our assembly structure in the ascent stage.

The first assembly constructed for the Ascent Stage was the Cabin Pressure Shell, which consisted of the Cabin Skin Assembly, the Front Face, and the Mid Section Assembly. For this example, we consider these leaf elements, but they were assembled largely from various components (such as bulkheads) machined specifically for the LM.

Once assembled, the Cabin Pressure Shell was joined by the Aft Equipment Bay Assembly (which we are also considering leaf). These two parts, once assembled together, became the Ascent Stage Structure.

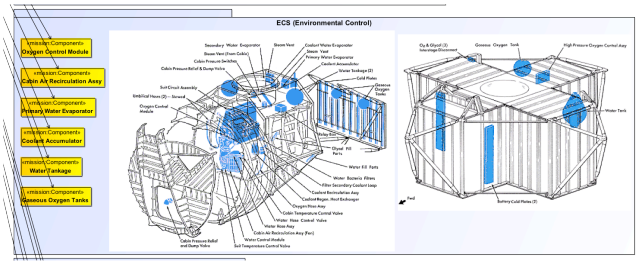




Once this core structure was complete, other components could be attached to ultimately create the complete Ascent Stage.

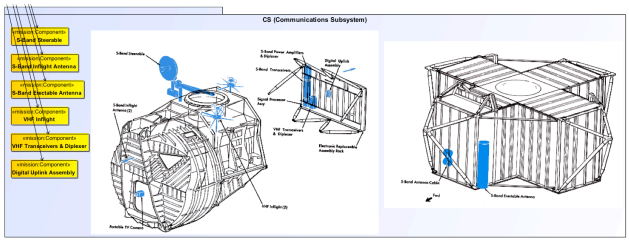
You will notice that the various children of the Ascent Stage are grouped within the same namespace. This is because these parts are identified as not only parts of the Ascent Stage in the assembly sense, but as existing as part of the realization of some functional performer (more on that later).

At left, select parts of the ascent stage that exist to help the LM perform its Reaction Control functions have been identified. The schematic at right shows where those parts that realized Reaction Control existed in the physical Ascent Stage assembly.



There is no reason that we have separated components into packages except for ease of understanding. As demonstrated by the existence of the schematics shown at left, some stakeholders were interested in having views which showed the realization of various concepts like Reaction Control and Environmental Control mapped into the assembly structure.

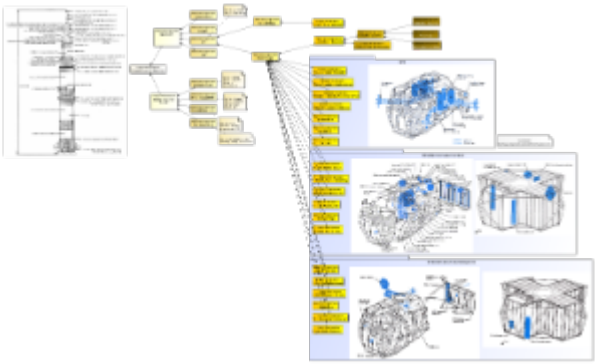
While this separation into packages seems like a convenient way to encode the information necessary to generate schematics like those shown at left, that is not how we would actually encode it; that pattern is covered in the Reconciliation/Abstraction Pattern. We chose to do this to make the modeling job easier and to create a more accessible view.



Another thing to note about these views is that some of these collections of components span multiple assemblies. For example, the ECS components and Communications components can be found in both the Ascent Stage and the Descent Stage. This illustrates the complexity and differences between the different kinds of trees; the assembly tree explains where and how components exist within the structure, but not why they exist or what they will do.

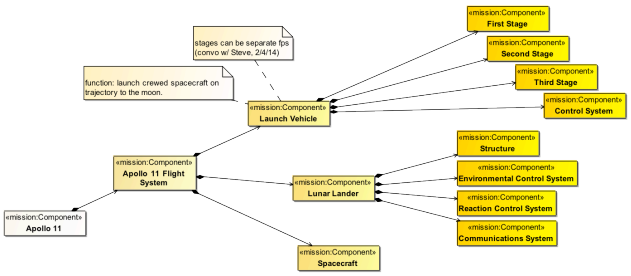
It also indicates that the mapping between a physical assembly and a functional performer is not always clean; it is often many to one, and the many may exist across several assemblies.

At right is the entire assembly diagram.



Functional Performer Space

The other kind of tree is the functional performer decomposition.



Here we take all of the Apollo 11 hardware (which could include ground systems), and focus on what we're calling the "Flight System". Starting from (slightly contrived for this pattern) first principles, if we say that the Apollo 11 mission must land at least one live human on the moon and return them safely to earth, we can assert that we will have a "Flight System" to make that happen.

Within this Flight System, we need to get the crew off earth, over to the moon, to the surface of the moon, back to the earth, and safely to the surface. We have decided to separate this problem into three pieces: a launch piece, a spacecraft, and a lunar lander. Obviously this is not the only possible logical decomposition, but we "did some trades" and determined this was the way to go forward (or, we know this is what happened).

Analysis of what it would take to get crew to the moon and back resulted in the responsibilities of the launch vehicle to be leaving the earth through injection of the spacecraft into a translunar trajectory. From there, the spacecraft would convey the crew and lunar lander to a lunar orbit. The lunar lander would deliver itself and its crew to the moon's surface, and back to lunar orbit, where it would rendezvous with the spacecraft, which would convey the crew back to earth.

This resulted in four elements composing the launch vehicle, each with different responsibilities. The first stage would provide initial speed, altitude, etc., the second stage would enter earth's orbit, and the third stage would set the spacecraft on its course to the moon. The control system would orchestrate the operation of these three stages in order to achieve the goals of the whole launch vehicle.

The lunar lander itself would have to provide a hospitable environment (or at least life-supporting) for the crew, communications with earth (including both receiving of various information and sending of information such as health of the spacecraft, videos, and biomedical telemetry). The lander also needed a means to adjust its movement in order to land safely and accurately on the moon, and a mechanical support structure to integrate, protect, etc. all of the elements.

This structure, while also a valid decomposition, has entirely different decomposition rules than the assembly structure.

Views

In order to keep the functional performer tree and the assembly tree distinct from each other, we chose to keep them in separate packages in the model. All elements and relationships belonging to a particular tree are contained transitively within the top package of each view.

Rules/Axioms/Invariants

ID	Restriction Rule
	Only «mission:Component»s, «mission:Item»s, and «mission:Flow»s can be Containers or ContainedElements.
	«mission:Component»s can only be composed of «mission:Component»s.
	«mission:Mission»s cannot be composed of «mission:Component»s.

	«mission:Item»s can only be composed of «mission:Item»s.
	«mission:Flow»s can only be composed of «mission:Flow»s.
	In both the functional performer and assembly structural decomposition, the relationship between elements is a composite association (black-diamond).
	The decomposition trees are directed and acyclic graphs.
	In a particular context, the relationship between ContainedElements and Containers is functional, i.e. each ContainedElement can have at most 1 Container.

Model Implementation Concerns

We briefly discussed the problem of "knowing the difference between the trees" or more formally, how to distinguish the taxonomy of tree types. Various solutions include stereotypes on packages which transitively contain the trees, put stereotypes on all of the relationships between the trees, stereotype the root node, put the tree in a container with a property specifying the type of tree. We are not ready to recommend any of these as the recommended practice for distinguishing between tree types. Most approaches offer either benefits in ease of modeling or ease of machine reasoning, but not both. Many approaches would allow a machine to transform the tree into something that could be more easily reasoned about, or transformed to keep consistency, or to provide more human-readable views. However, further investigation (through user feedback) must be conducted before any approach can be recommended.

Validation/Well-Formedness Reasoning

We propose two general validation rules:

- Elements in the tree should be of the same "type"as described earlier. The approach for asserting and checking the type, at this point, is left to the modeler.
- Each tree should be acyclic and have a single parent.

Supporting Scripts/Tooling

Name	Description	Status	Author/Provider	Location
Tree Comparator	Given root nodes of two decomposition trees, checks that the leaf elements match in both trees; identifies missing components, etc.	Completed for preliminary Flight System modeling on InSight; some assumptions might be slightly out of date	mjackson	*CompareMassTrees.py* *(Stash)*
InSight Model Helpers	Other generic model helpers to simplify basic MD Ui tree tasks such as changing navigability.		mjackson	
Systems Reasoner	May contain useful python libraries and examples		??	Comes bundled with SSCAE MagicDraw
MEL Plugin	DocGen extension that creates Bill of Materials and Deployment tables.	Currently implemented as DocGen java extension		
Super MEL	Does even more MEL stuff!		Bjorn Cole	SuperMEL Community Page

Scripts often contain inherent assumptions about the details of the tree structure, and should ask "is this pattern/model well formed for purpose of this script?"

When writing a reasoner like this, know that validation can be expensive. Possible recommendation is to write script assuming validation has succeeded... or to include validation on execution.

Tooling Tricks

Trick	How to do it	Why?

The intended result of the trick	*Explain (text, screenshots if necessary) how to repeat the trick*	*Explain why this is important. If the trick enables reusability across teamwork modules, explain how. If the trick makes implementing the pattern faster, explain.*

Open Questions

Question	Status	Discussion
Should each type of tree have its own specialization of aggregation?	OPEN	
Correspondence between trees?	OPEN	Reconciliation/Abstraction Pattern
What exactly is the embedding of this "View" that determines the type of decomposition / the exactly implication of the part/whole relationships used?	OPEN	discussed previously?

Advanced Questions

How do I tell what IS and what is NOT structural decomposition?

References and Pattern Resources

Six Different Kinds of Composition - Odell

List any relevant papers, presentations, links, references, etc.

Further Examples

List or link to any examples that are relevant and approved by the pattern developer.

Community Page

Link to Community Page

Copyright