Ontologies and Diagrams for Software and Systems Engineering: processes and objects

Roel Wieringa*

January 23, 2000

1 Introduction

Structured analysis, which arose in the 1970s and was popular till the end of the 1980s, raises the principles of structured programming to the level of systems, which may consist of software, hardware, and people. The principle of functional decomposition is common to structured programming and systems engineering and is therefore a central theme of structured analysis. The emphasis on the interaction of processes that creates emergent behavior in the system was imported from systems engineering. This matched nicely with the emphasis on implementation-independence and problem-orientation, inherited from structured programming. Adoption of the discipline of allocation and flowdown of system functionality to subsystems, central to systems engineering, was less successful.

Since the beginning of the 1990s, structured analysis has lost, and object-oriented analysis has gained in popularity in software engineering. But history only partly repeated itself. Object-oriented analysis does not raise the principles of object-oriented programming to the systems level but remains at the level of object-oriented programs. And the object-oriented design principle, which, as explained below, is subject-domain-oriented, does not match with a design principle from systems engineering. Nevertheless, the idea of an entity with local state and behavior that emerges from the collaboration of lower-level entities is shared by object-oriented analysis and systems engineering. And the allocation and flowdown of functionality from the system to its component software objects is more successfully adopted in object-oriented than in structured development.

In this paper, I investigate the possibility of merging ideas from structured and object-oriented analysis to get a blend that fits well into a systems engineering approach. To this end, I first briefly present in section 2 a framework for software and systems engineering, that consists of a number of design dimensions. In section 3 I show how the diagram techniques from Yourdon structured analysis and the UML fit into this framework. Next, I discuss in section 4 the ontologies presupposed by these diagrams. This leads to several different possible mergers of the diagrams from the Yourdon and UML approaches. Section 5 summarizes the results and discusses some open issues.

*Department of Computer Science, University of Twente, P.O. Box 217, 7500 AE The Netherlands. Email: roedw@cs.utwente.nl. http://www.cs.utwente.nl/~roedw.
2 A framework for software and systems engineering

Detailed analysis of six structured and 19 object-oriented methodologies [15], as well as an analysis of frameworks for product engineering and systems engineering [1, 4, 6, 10, 14], has led to the following simple framework for software and systems engineering.

- Process dimensions
  - Temporal dimension: Design strategies.
  - Logical dimension: Justification of design decisions.

- Product dimensions
  - Requirements dimension: Desired functionality, behavior, and communication of system.
  - Architecture dimension: Classification and multiplicity of system parts.

The two process dimensions concern the strategy by which a design was found (e.g., incremental or concurrent) and the argument by which the design is justified (problem analysis, alternatives considered and their estimated problem-solving power). The process dimensions are not the topic of this paper.

The two product dimensions are requirements and architecture. Requirements are desired properties of a system. These include at least the desired system functions and for most software systems, two aspects of system functions are specified: behavior, which is the ordering of functions in time, and communication, which is the ordering of functions in “space”. A system function is any piece of external system interaction that delivers a value to its environment. System behavior is the structure of the possible occurrences of system functions over time; it may be represented by stimulus–response pairs, by finite-state transition diagrams, by differential equations, etc. Communication is the transfer of matter, energy or information between a system and its environment when the system is exercising a function.

The architecture dimension consists minimally of an indication of the types of parts that make up the system and of their multiplicity. The absolute multiplicity of a class is a set of natural numbers that tells us how many instances of this class there can be at any point in time. The relative multiplicity of a class with respect to another class is an indication of how many instances of one class there can be for every existing instance of the other. For systems with a dynamically growing and shrinking number of parts, such as software systems, multiplicity is an important architectural characteristic.

The two product dimensions are recursive: Each part itself has requirements and an architecture. Since the communication properties of each part tells us how it is connected to other parts, architecture models usually contain a representation of the communication structure of all parts of the system in addition to the classification and multiplicity of the parts.

In previous research, I showed that all techniques offered by software engineering methods can be classified along these dimensions [15]. Observe that the framework makes no difference between software systems and other kinds of systems. The reason is that differences between software and non-software systems do not influence the framework. The differences have their root in the fact that a computer is a programmable machine. A program is a sequence of instructions that tells a computer how to simulate machine. This leads
to two important differences. First, a program is a collection of symbols. This means that
(1) software is invisible and (2) seems to be easily changeable, and that building a software
system consists of producing text, just as design does. This leads to a curious inversion of
terminology, in which what is called design elsewhere (a statement of what you are going
to build) is called requirements in software engineering, and what is called a specification
elsewhere (a parts explosion) is called a design in software engineering.

The second difference is that the simulated machine is a symbol-manipulating machine.
This means that the simulated machine, and hence the computer when executing the pro-
gram, communicates with its environment by exchanging symbols. Symbols are not given by
nature but defined by people. Each symbol has a meaning, which is defined in a dictionary.
There are rules for combining symbols and for assigning meaning to structured symbols.
The subject domain of a software system is the part of the environment referred to by
the atomic and structured symbols exchanged with the environment. To understand the
communication between a software system and its environment, one must understand the
subject domain. A software requirements specification usually presupposes a model of the
subject domain. Since the subject domain itself is a system, this can consist of the same elements
as a model of the software system: an indication of the classification and multiplicities
of the parts of the subject domain and a description of the behavior and communication of
the parts. The same framework that I apply to software systems is applicable to the subject
domain.

3 Diagram Techniques

Table 1 lists all diagram techniques in the Yourdon Systems Method (YSM) [16] and the
Unified Modeling Language (UML) [13], classified along the product dimensions of my
framework. The table illustrates that my framework is useful to group techniques in such a
manner that it might be possible to freely combine techniques from different groups and to
merge techniques within one group. The following discussion assesses the extent to which
this is possible.

Classification and multiplicity. The YSM entity-relationship diagram (ERD) can be
used to specify the types of entities in the subject domain and their absolute and relative
multiplicities. The IEEE recommended practice for software design specifications [8] uses
ERDs to represent entities in a software architecture. The UML static structure diagram
(SSD) represents the software objects that make up the system and, sometimes, the entities in
the immediate environment of the system. It is easy to view the ERD technique as a
subset of the SSD technique and use the combined to represent the classification and
multiplicity for objects in the software system, entities in its immediate environment and
entities in the subject domain. The resulting technique is able to represent a collection
of types (classes) or discrete entities (objects) that can have a local state, represented by
attribute values. In addition, it can represent objects have behavior encapsulated in them
and present an interface to their environment. There is a large number of special annotations
and adornments defined for SSDs, that are meaningful only when they are used to represent
the structure of programs. If an SSD is used for a different purpose, then these annotations
and adornments cannot be used.

UML component diagrams are used to declare the executables, sources, libraries and
<table>
<thead>
<tr>
<th>Classification and multiplicity</th>
<th>Functions</th>
<th>Behavior</th>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>YSM entity-relationship diagram</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UML static structure diagram</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>UML component diagram</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>YSM mission statement</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function refinement tree</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>YSM event list</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>YSM Mealy diagram</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>UML statechart</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UML activity diagram</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>YSM data flow diagram</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Communication diagram</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UML deployment diagram</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>UML use case diagram</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>UML collaboration diagram</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>UML sequence diagram</td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: YSM and UML techniques and the aspects they represent.

other components from which a software system can be built, and to represent dependencies between them. They represent classes of things but they do not represent their multiplicities. The use of component diagrams is restricted to represent components at the implementation platform level, but with this restriction, a component diagram can be viewed as a specialization of an SSD.

**Functions** The YSM mission statement specifies the top level functions of a system. The function refinement tree is a simple technique well-known from various branches of systems engineering. It can be used to expand the system mission to a list of functions, ordered in a hierarchical structure. Such a tree does not represent the architecture of the system but is merely a logical organization of external functionality. It shows us why a certain function must be present: To achieve a higher-level function, and ultimately to achieve the system mission.

**Behavior** The basic representation of behavior is the event list. In its simplest form, this is a list of all possible system stimuli and for each stimulus, the desired response(s) of the system. A response can be described as a dialog between the system and its environment, referring only to the meaning of this dialog for the environment. In that case, the responses
are what the UML calls *use cases*. Alternatively, we can refine the responses until they are atomic and incorporate the possible system states in the list, in which case the event list turns into a state transition table for the system. A finite state transition table can always be represented by a state transition diagram (STD), which represents its structure visually—although one may need a very large sheet of paper to do this. Three widely used conventions for STDs are the Mealy diagram, used in YSM, the Moore diagram, used in the Shlaer–Mellor method [12], and the statechart, used in Statemate and the UML. Statemate [7] is a structured analysis approach that uses one flavor. The syntax and semantics of statecharts in Statemate and the UML differ.

Event lists and various kinds of state transition diagrams can be used to represent system behavior or the behavior of parts (e.g., of software objects). In structured analysis, they are used to express the sequencing of processes. In most object-oriented methods, they are used to represent the behavior of software objects.

The event list, Mealy diagrams, Moore diagrams and statecharts are alternative ways to represent behavior. The UML activity diagram is something different. It represents behavior in the classical flowchart way extended with constructs for parallelism, the flow of objects between activities, and the allocation of tasks to actors. Activity diagrams do not combine well with the other techniques listed in the table and they are best used only for the representation of workflows in organizations.

**Communication** The YSM data flow diagram (DFD) shows how the processes and stores that make up the system and the external entities outside the system send data and events to each other. The communication diagram is a technique from the Shlaer–Mellor method [12], not present in the UML. It shows, at the class level, how objects are connected by channels through which they can send messages. The UML deployment diagram shows how resources—basically anything that can do a computation—are connected by channels, without showing what kind of information passes through these channels. The UML use case diagram shows how the system is connected to external actors by (logical) channels without showing what information passes through these channels. It is a hybrid technique, comparable to a simplified top-level DFD, that shows the use cases of the system and the actors that interact with the system during a use case.

UML collaboration and sequence diagrams differ from all other diagrams because they are used to *illustrate* communication sequences rather than to specify it. Damm and Haré [5] propose an extension of sequence diagram for specification to make them fit for use in system specification, that should cover all possible communication sequences rather than individual instances of it. We have to wait and see whether this will be adopted by the OMG in a future version of the UML.

There are numerous differences between the ways communication is specified in YSM and the UML. An important issue is the way a sender addresses the destination of the receiver(s) of a communication: In DFDs, the sender addresses the channel and does not know which receiver(s) are connected at the other end. In the UML, the sender addresses the receiver(s) and does not know the channel by which the receivers are reached. Other differences concern the use of synchronous or asynchronous communication, the presence of delay between sending and receiving, and the speed with which the receivers process the communication. These issues are subject of current research.
4 Architectural Ontologies

More fundamental even for an assessment of the possibility to merge the notations is the architectural ontology presupposed by YSM and UML notations. An ontology is a metaclassification of kinds of entities that we deal with.\(^1\) An architectural ontology is the ontology of a class of system architectures. Analysis of DFDs in many YSM models reveals that there are two kinds of processes and two kinds of stores. One kind of process is the stateless transformation, which is a mathematical function that when prompted, computes an output from the available input. The second kind of process has a state and persist over time. When prompted, they transform input into output in a way that depends upon their local state, which may evolve over time. This kind of process is really an object in the object-oriented sense of the term.

Stores are elements that remember information written to them until it is deleted. In the meantime the information can be read and updated. Now, given a set \(D\) of data types, a variable is a store that at any one time contains exactly one value of a data type from \(D\). The data type can be atomic (e.g. integer) or structured (e.g. a record structure). By contrast, a database is a store that at any one time contains a finite set of values of a data type from \(D\). For example, it contains a set of records that contain personnel data. The distinction is relative to the given set \(D\) of data types. For example, if Person is one of the data types, a store that contains a set of Person instances is a database. By adding the structured data type set of person to \(D\), we turn the store into a set-valued variable.

The technique to represent classification and multiplicity of parts is the static structure diagrams, which includes the techniques previously called class diagram and object diagram [3, 11, 13]. Here, we find a simpler ontology:

- An object is a discrete entity with local state and behavior and an interface to other objects.
- A class is a set of objects with similar properties. A class is a type-level concept, so a class is really a set of all possible objects that share a given set of properties. These objects are called its possible instances.

There is no barrier to merging these ontologies. We can make a model of the environment or of the system architecture that contains all of these kinds of entities. Different mergers of these ontologies require different diagram techniques to represent them. In the following, I describe three possibilities.

4.1 The techniques in TRADE

The first option is to keep it simple and stick to an object-oriented ontology, which fits well with a systems engineering approach anyway. We can then choose a minimal set of techniques for each aspect—at least one, at most two—and define consistency rules for these techniques when they are used in one model. Table 2 shows one such combination, that I call TRADE (Techniques for requirements and design Engineering).

The techniques in TRADE can be used at different levels in the system hierarchy: for the entire environment with the system in it, for the system alone, or for any part of the system. At each of these levels, the outline consistency constraints are as follows.

---

\(^1\)In some circles the term “ontology” is used in the meaning of “reference model of the subject domain”. My use of the term is more general and stays close to the original meaning of the term in philosophy.
<table>
<thead>
<tr>
<th>Static structure diagram</th>
<th>X</th>
<th></th>
<th>X</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mission statement</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function refinement tree</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Event list</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Statechart</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communication diagram</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: The specification techniques in TRADE.

- For each class in a static structure diagram there must be an event list or statechart that defines the behavior of the instances of the class.
- The stimuli and responses in the event list or statechart of a class must be operations or signals of the class.
- The classes in the static structure diagram must be the same as the classes in the communication diagram.
- The interface of the classes in the communication diagram must agree with the interface as declared in the static structure diagram (and therefore with the interface of each event list and statechart).
- If there is a mission statement and a function refinement tree of a class, the root of the tree represents the mission of each instance of the class and the children of the root represent responsibilities of the instance. (In an environment model, class instances may be entities, observers, actors, and the system itself. In a system model, class instances are software objects.)
- If the responsibilities of a class are declared in the class diagram, these must agree with the responsibilities listed in the function refinement tree of the class.

To make this more precise, the syntax and semantics of all diagrams must be specified precisely, and the constraint reformulated in terms of this syntax and motivated in terms of this semantics. This goes beyond the limitations of this paper.

The relationships between the model of the environment and the model of the system are traditionally called traceability relationships and can be represented by a traceability table. This can be viewed as a database table, that if it is very small, can be visually represented visually by a table. We can define such a table for different aspects of the system, that shows how external functions are allocated and flow down to functions of parts, and similarly for stimulus-response pairs and external communication. This is a central technique in systems engineering.
4.2 Augmented data flow diagrams

Rather than an object-oriented ontology, we can choose a process-oriented ontology. This requires us to replace the communication diagram by a DFD. Since this is an instance-level technique, the SSD is not needed for representing the system architecture anymore—the DFD already represents it—but we may still wish to use it for describing the architecture of the environment.

We can simplify the representation of the process-oriented ontology, by using extended state transition diagrams (STDs.) An extended STD is an STD with local variables, that can be tested and updated along the transitions. Extended STDs are used the telecommunications specification language SDL [2]. An STD without local variables has a finite state space. Each local variable extends the state space by multiplying the number of possible states with the number of possible values of the variable. If there is a local variable with an infinite type, such as Integer, then the state space is infinite. The price we must pay for this increase in expressive power is a lack of visual representation of the structure of the extended state space, and an added design choice: Should we represent an aspect of the state by a node in the STD or by a value of a variable? UML statecharts are extended state machines, in which the object attributes are the local variables.

All structured analysis methods except Statemate uses non-extended state machines to represent behavior. This means that any test or update to be performed by a state machine has to be performed by a stateless transformation. Figure 1 illustrates this. Because the Mealy diagram has no local variables and can perform no computation, it needs a data process to do a test and then enters a decision state, that it will immediately leave when receiving the answer. The figure also illustrates that if an action contains a computation, a state machine without local variables must trigger a data process to do the computation for it. This situation is unaltered if we would use statecharts rather than Mealy diagrams. The important point is the presence or absence of local variables in the STD.

![A data flow diagram.](image1)

![A Mealy machine for the control process.](image2)

Figure 1: A non-augmented DFD with a non-extended Mealy diagram for the control process.

An augmented DFD is a DFD with extended control processes, and an extended control
process is a control process specified by an extended STD. The extended control process in
figure 2(a) is specified by the extended Mealy machine of figure 2(b), that has local variable
V. An extended control process may have data flows as input and output. Input and output
data flows must be declared as variables. A variable not occurring as input or output data
flow acts as local memory of the control process. Transitions can now include a test in their
Guard and can include update actions in their transitions, allowing us to drop the data
processes needed before. This considerably simplifies the DFD. The processes that remain
in the DFD are either objects or stateless transformations needed for another purpose than
updating a local variable, such as for example accepting input or producing output.

4.3 Augmented static structure diagrams

If we want to retain the type-instance distinction characteristic of SSDs, we can lump
together the process-oriented and object-oriented ontologies and augment SSDs to represent
this. An augmented SSD is an SSD extended with elements that represent transformations
and stores. Given the set of data types and object classes in an SSD, some of the stores
will be variables and others will be databases.

An obvious visual representation of transformations and stores is by means of the pro-
cess and data store symbols from DFDs. These can be easily accommodated to SSDs by
the stereotype mechanism. However, we can reduce the number of icons in augmented
SSDs by encapsulating transformations into utility classes stereotyped “Transformations”
and encapsulating variables into utility classes stereotyped “Variables”. Utility classes are
standard UML construct. This leaves us with databases as the only non-object-oriented
element visible in an augmented SSD. One often sees this in client/server architectures. At
the lowest layer, there are one or more databases that contain business data. In the level
above, a number of business objects encapsulate data from the database with operations
and signals that represent business logic. In the level above that, user interface objects take
care of the interaction between users and the business objects. Supporting all of this, there
may be utility classes containing global variables or transformations.
Augmented SSDs allow is to represent the different views of the type-instance distinction in process-oriented and object-oriented ontologies in one diagram. A class C encapsulating attributes a and b and operations O1 and O2 can be represented in the same diagram by a database C instances, containing tuples (a, b), accessed by stateless transformations O1 and O2.

5 Discussion and conclusions

To summarize, the ontologies of structured and object-oriented analysis can be merged into an ontology of classes, objects, stateless transformations, variables and databases. The diagram techniques can be merged such that the resulting combination matches the chosen ontology and contains techniques for the aspects of functionality, behavior, communication and architecture. The resulting design approach can be embedded in a systems engineering process with due attention for the role of the system in its environment, the emergence of system behavior from the interaction of its parts, and proper documentation of allocation and flowdown of functions from the system to its parts.

I have not discussed the decomposition principles commonly used in structured and object-oriented analysis. Briefly, there are three kinds of decomposition.

- **Functional decomposition**, used in structured analysis and systems engineering, defines one system part for each external system function.
- **Subject-domain-oriented decomposition** defines one system part for every significant subject domain entity (used in object-oriented methods) or for every significant event in the subject domain (event partitioning, used in structured analysis).
- **Stimulus-response chain partitioning** defines one system part for every stimulus-response chain. This is used in structured analysis (combined with event partitioning), object-oriented analysis (using collaboration or sequence diagrams to represent the chain) and systems engineering. At the level of organizations, this kind of partitioning is called “rotating” the organization and involves the introduction of customer-oriented units.

These decomposition principles can be combined in the design of one system. They can also be combined with different architectural ontologies.

Another issue not touched upon in this paper is the various architectural levels relevant for software engineering. There are at least four:

- Ignored in structured and object-oriented methods is the **external architecture**, which is the architecture of the environment in which the system is to be embedded. An understanding of the external architecture is needed to understand the system requirements.
- Structured analysis always works on the **essential level**, at which perfect implementation architecture is assumed [9]. An essential architecture is motivated only by the requirements and the constraints from the usage environment.
- The UML is currently being defined at the **implementation level**, where we take the relevant properties of the implementation platform into account, such as finite speed and finite processing resources.
The fourth relevant level is the implementation platform itself, consisting of a network of computing resources that offer a particular infrastructure. This is represented in the UML by a deployment diagram.

Design guidelines and traceability relationships must be refined if we take these different architectural levels into account. Current research includes a detailed analysis of these levels and of the relationship between design patterns that can be found at these levels.

References


