APPENDIX 3

Description of

for Software Design Minimal Methods

STATE MACHINE METHOD

DESCRIPTION:

The State Machine method is fundamental to understanding all software systems. It is the canonical way to represent software and other types of systems. It represents the reactive ability of the system to stimuli.

CONCEPTS:

Tutorial:

The State Machine is sometimes called Finite State Machines (FSM), Finite Automata, or just AUTOMATA. It is a control structure that associates an ACTION with an input EVENT and a AUTOMATA STATE. When the automata reacts to an input it resets its own new state. All computing structures may be reduced to an automata or an automata with a pushdown stack which is equivalent to a Turing Machine.

For actual use an automata needs to allow nested states and have modes. Nested states, or sub-automata allow for a variety of responses depending on the situation as do modes. In fact a mode is normally a sub-state of a higher level state machine.

The automata is the dual of the PetriNet. The PetriNet displays control explicitly where as the automata displays control implicitly. This is to say that in a state machine the control structure is coded into tables of stateVectors where as it is a direct link from one design element to the next. Sometimes the control path of the system may be very involved and may jump from state to state in a way that is difficult to understand. In those cases the number of PetriNets needed to represent a single state machine would be very large. In fact there is normally a combinatorial explosion between the representation of a system as a PetriNet rather than an Automata.

Principles:

1. Discrimination of System States into a finite set.

2. Association of Actions with Events and the System State.

3. The System may only be in one state at a time.

Properties:

1. Transitions connect two states and are associated with an event and action.

2. An automata may have multiple modes or meta-states.

3. Each state may be decomposed into sub-states and be associated with a lower level automata.

4. Automata may appear in the CONTROLSPEC of the Controlflow Method flanked by decisionTables and processActivationTables.

Criteria:

1. Automata should be constructed to be as compact and simple as possible.

2. Unnecessary states should be eliminated.

3. It is better to have multiple small state machines than one large state machine.

4. System modes should be minimized. Normally there is a Error or Fault Handling modes, Initialization mode, and normal running mode.

Guidelines:

1. Each entity within the system may have its own special lifecycle which should be modeled with an Automata.

2. Automata at different levels in the hierarchy need to interact properly and special care should be taken to design and test that interaction.

Measures:

1. Number of Automata

2. Number of States and Events per Automata

3. Number of Modes

Notes:

1. See STATEMATE from ILOGICS for an example of a design method based on embedded state machines.

ACTIONS:

1. Define input events.

2. Define output actions.

3. Define system states.

4. Define sub-states.

5. Define Modes.

6. Connect states by transitions.

7. Connect Actions and Events to transitions.

REPRESENTATION:

Elements:

1. Current State of the Automata.

2. State

3. Event

4. Action
5. Mode

Relations:

1. Transition from State to State
2. Between Event, Action, and Transition
3. Between Automata and sub-Automata.

Notation:

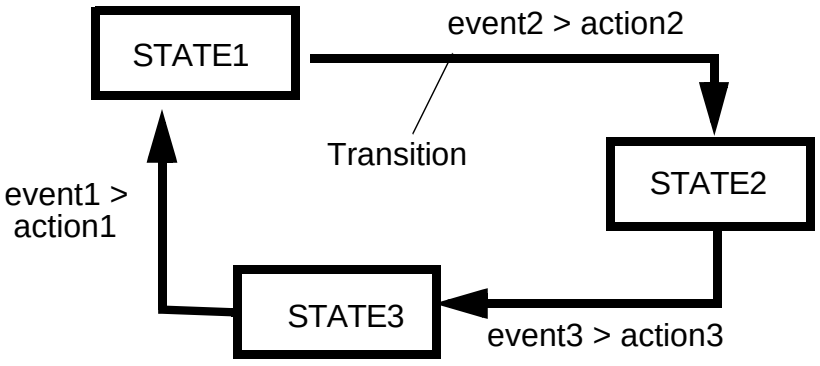
1. There are two different state machine notations. In one actions happen along the transitions where as in the other actions happen on entering a state. We have selected the notation where the actions happen along the transition.

Artifacts:

1. Automata may be represented as Tables or graphically as state diagrams.

EXAMPLE:

1. The structure of an automata.



PETRINET METHOD

DESCRIPTION:

The Petri Net is a formalism that shows control flow through the system. Instead of being reactive like the state machine it is self-activating. It replaces the old method of using flowcharts to understand system control flow. Colored Petrinets allow a great deal of diversity in the simulation of active control structures.

CONCEPTS:

Tutorial:

Petrinets are networks connecting PLACEs and TRANSITs. Each transit has a set of input places and a set of output places. Tokens or markers move through the places. When the correct configuration of markers appear in the input places as determined by a DECISION rule then the transit will fire and place an arrangement of tokens in the output places. In the original petrinet all the tokens were the same. In the more modern version called colored petrinets different kinds of tokens may flow through the network. A transit may react differently to different sets of tokens in its input places. When the transit fires it is simulating the accomplishment of functional work.

Principles:

1. Control flow lines are distinguished explicitly instead of implicitly.

2. The system is self activating.

Properties:

1. Places are connected to transits.

2. Colored markers move through the network of places and transits.

3. Rules govern the firing of the transit.

4. When the transit fires functional work is accomplished.

5. A transit may contain a sub-Petrinet.

6. The petrinet transit may be associated with a variable called a Banner whose contents is called an Insignia. When the transit fires it may set the banner and further firing many depend on the value of the banner.

Criteria:

1. Petrinets should only be used when control flow is problematic and needs to be studied, otherwise automata are preferred because it only takes one automata to represent the same data as many petrinets.

2. Transit decision rules should be minimized.

3. The number of colored markers should be minimized.

Guidelines:

1. Keep petrinets as simple as possible and use sub-nets to hide detail.

Measures:

1. Number of places and transits in a net.
2. Number of sub-nets

Notes:

1. See the book on Petrinets by Peterson.

ACTIONS:

1. Define Places
2. Define Transits
3. Define Banners
4. Define Decision rules.
5. Link places to transits.
6. Decide on starting positions for tokens.
7. Link tansits to functions.

REPRESENTATION:

Elements:

1. Place
2. Transit
3. Banner

Relations:

1. Banner to transit
2. Decision rule to transit
3. Place to Transit

Notation:

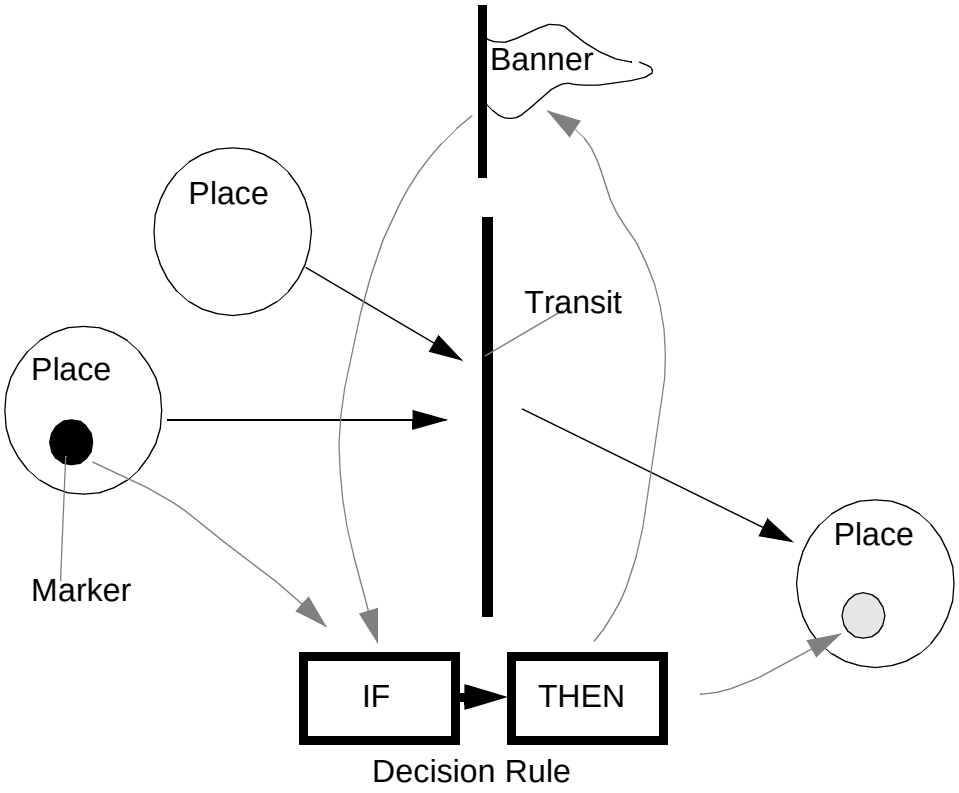
1. Petrinets are normally drawn but they may be represented as a matrix of places by transits.

Artifacts:

1. Petri net diagrams or tables.

EXAMPLE:

1. Structure of Colored Petrinet



DATAFLOW DECOMPOSITION METHOD

DESCRIPTION:

The dataflow method is a means of functionally decomposing a system which takes into account the relations of functions to data. It is a two way bridge between the data and functional views of a software system. Functionality concerns what the system does and data concerns where the system does it in memory. The dataflow diagram allows one to determine what is done to data and how data connects to what is done in the system in general.

CONCEPTS:

Tutorial:

The dataflow method is used to get a coherent picture of the functionality of the software system as a whole. The dataflow method is employed in either requirements analysis or the early stages of software design. The model used in requirements analysis is called the essential model. It is employed to get a functional view of the entire design space without implementation constraints. In the design process these implementation constraints are introduced and the essential model is transformed into the implementation model that allows the design to be traced back to the requirements at every stage of the design.

One always begins the functional decomposition of the system by identifying the externals of the system and thus the system boundary. The context diagram is the end result of this analysis. Then the context diagram is successively decomposed into lower and lower levels of functionality until the system is sufficiently defined. That definition is done in terms of the data interfaces of functional subsets of the system. The identification of the data inputs and outputs is the major means of defining the functionality of a particular part of the system. Data decomposition goes hand in hand with functional decomposition. The decomposed data is kept in a data dictionary where sub-relations between data-items is maintained along with data definitions. data-stores are identified where groups of persistent data-items are kept. Some attempt to use normal form database methods should be applied to defining data-stores.

Principles:

1. Top-down decomposition of both functions and data.

2. Identification of the external interface with a context bubble.

3. Definition of functional and data components by identifying inputs and outputs.

4. Functions are separated by the kind/type relations: i.e. similar functions are grouped together and different kinds of actions are separated systematically by the functional decomposition process.

5. A firm distinction between essential and implementation dataflow models.

Properties:

1. The data flow diagram is a series of nested networks of dataflow bubbles, externals, and data-stores connected by dataflow arcs.

2. Any dataflow bubble can be decomposed into a lower level dataflow.

3. Any data-store or dataflow arc can be decomposed into data subcomponents

4. Dataflow bubbles, arcs, and data-stores are strictly hierarchical.

5. The essential dataflow model does not contain any implementation constraints and assumes perfect or ideal conditions like infinite speed, infinite storage capacity, perfect transmission along datalines, etc.

6. The implementation dataflow model is impure in that it contains alterations that take into account implementation constraints.

7. The dataflow diagrams of the system is accompanied by the data dictionary which describes all the data relations in the system.

Criteria:

1. The dataflow representation of the system should all be developed to the same level of detail.

2. No agent or event characteristics of the system should appear in the dataflow diagrams.

3. A given dataflow bubble should contain no more that nine sub-bubbles

4. There is no stopping rule for deciding when the dataflow decomposition is finished. Normally the first few levels, say at least three, should be done for every system. Too many levels of decomposition quickly becomes unwieldy.

5. The dataflow is meant to give a clear idea of the functional design space as a whole. Once this objective is achieved then further work on this system view is at the discretion of the designer.

Guidelines:

1. The dataflow may either represent the existing system or the proposed system.

2. The dataflow identifies the system boundary and represents the whole system.

3. The essential model should be developed first so that first versions should not contain implementation constraints details.

4. The dataflow essential model is used as a test for the coherence of system requirements.

5. Data items should as much as possible be grouped into tables to prevent unnecessary data expansion in the design of the system.

Measures:

1. Number of levels in functional decomposition.

2. Number of data-items.

Notes:

ACTIONS:

1. Functional decomposition

2. Interface Identification of inputs and outputs.
3. Identification of external entities
4. Data decomposition
5. Leveling the hierarchy of dataflows.
6. Following datalines to assure path coherence.

REPRESENTATION:

Elements:

1. Dataflow bubble
    - Name
    - Indented Number
    - Sub-bubbles
2. Dataflow arc
    - Name
    - Sub-data-items
3. data-store
    - Name
    - Sub-data-items
4. External
    - Name
    - Sub-externals

Relations:

1. Dataflow bubbles have input and output data-arcs.
2. data-stores have input and output data-arcs.
3. Externals have input and output data-arcs.
4. Dataflow bubbles contain sub-bubbles
5. data-stores contain data-items
6. Data arcs are data-items
7. Externals are only outside the context bubble.

8. All other bubbles are within the context bubble.
9. Externals to any particular lower level dataflow may or may not be shown.
10. Externals may or may not be decomposed.

Notation:

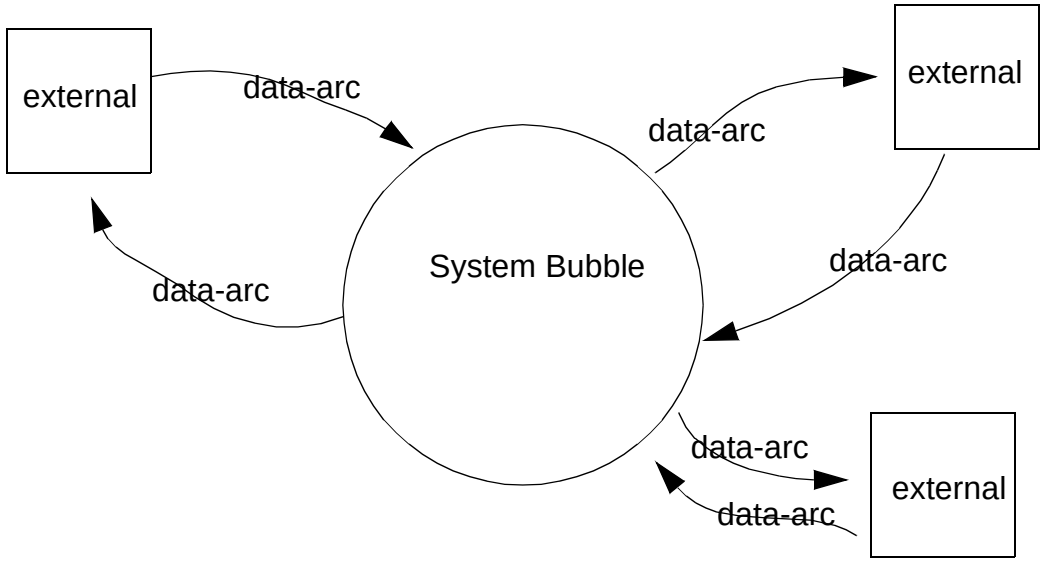
1. Yordon/DeMarco notation
2. Gane/Sarenson notation
3. Hatley/Pribahi notation

Artifacts:

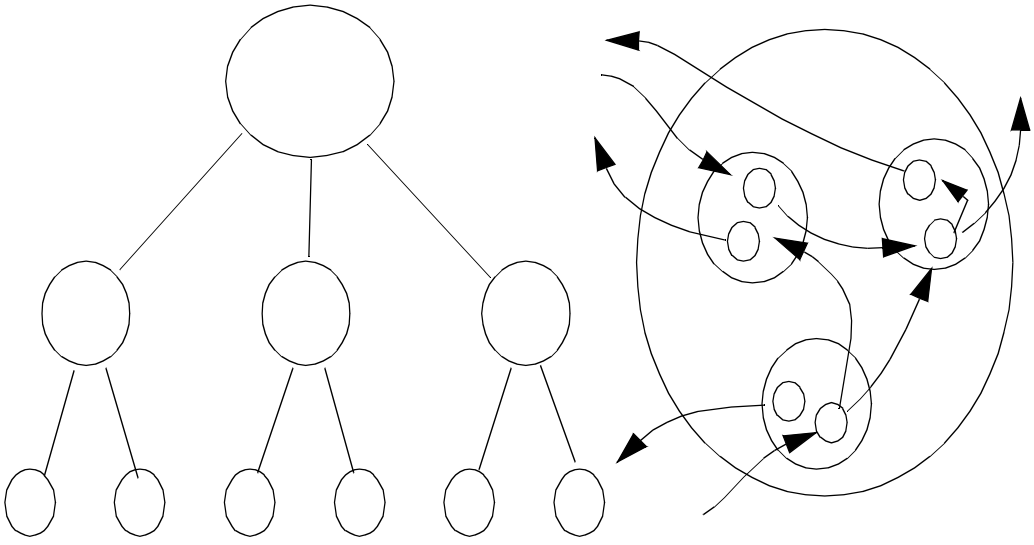
1. Context diagram
2. Hierarchically ordered dataflow diagrams
3. Data dictionary
4. Functional descriptions

EXAMPLE:

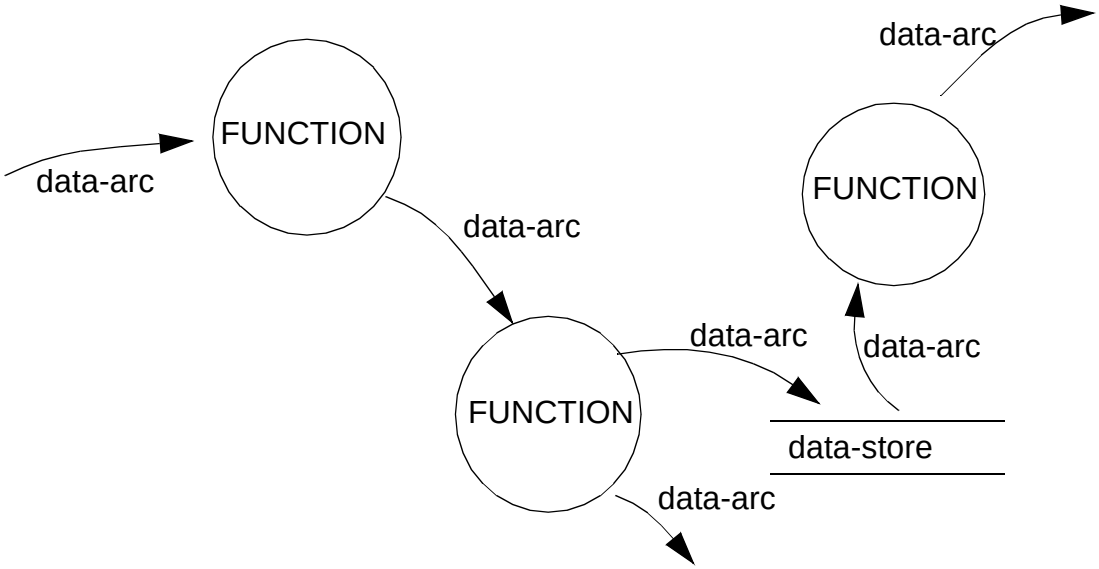
1. Context Diagram



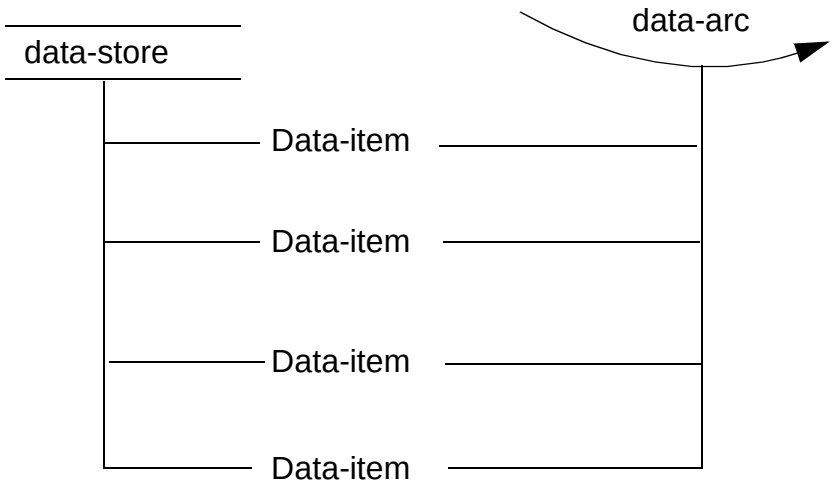
2. Functional Decomposition



3. Dataflow with data-store



4. data-store and data-item decomposition



CONTROLFLOW DECOMPOSITION METHOD

DESCRIPTION:

The controlflow is an addition to the dataflow diagram which makes explicit control information missing from the dataflow diagrams as originally conceived. This addition was made in order to adapt the dataflow method to the functional analysis of real-time systems.

CONCEPTS:

Tutorial:

Principles:

1. Control information flows through software systems along lies parallel to but distinct from data.

2. Control information may originate from functional bubbles, externally or from datastores and flows to control specifications.

3. Control information flows from control specifications to activate functional bubbles.

4. Control may appear as interrupting triggers that make things happen in the system.

Properties:

1. Control diagrams may or may not have dataflow lines on them.

2. Control lines always flow to control specs on the same diagram never between dataflow diagrams. This means that control is always one level higher in the hierarchy of functional decomposition than what is being controlled.

3. Control specs may contain a variety of control structures such as state machines, process activation tables, decision tables, petri nets, etc.

4. Dataflows fire when all their data inputs are available unless they are deactivated by a process activation table in which case it is as if they were taken out of the system all together.

5. Thus control exerts what is really equivalent to modal changes in the system. Specific control sequences are not modeled with control flows but should instead be modeled with petri nets or state machines.

6. Triggers may be introduced to show response to external stimuli (this is a weakness of the Hatley Pirbhai model and is taken from ESML)

Criteria:

1. Control should always be minimized. No control flows should appear where a dataflow line would do.

2. The distinction between data and control is hazy and so control should only appear where it is clearly called for and is definitely distinct from dataflow.

3. Only as many control flow and control specs should be introduced as necessary to specify all the

modes in the system.

Guidelines:

1. Control specs may connect different levels of the dataflow diagram but as much as possible control should be limited to the level at which the controlspec resides.

2. Triggers should be avoided unless absolutely necessary as these are a deviation from the Hatley/ Pribhai methodology.

Measures:

1. Number of control specs.

2. Number of triggers.

Notes:

ACTIONS:

1. Define control lines for a particular dataflow diagram and represent them going to the controlspec.

2. Identify all triggers within the system.

3. Design the control structure within the control specification.

4. Design the relation of the control structure back to the processes using a process activation scheme.

5. Make sure the modalities of the system are coherent.

REPRESENTATION:

Elements:

1. Control arc

2. Controlspec

3. Decision Table

4. State Machine

5. Petri net

6. Process activation table.

7. Triggers

Relations:

1. The control arc goes from a data bubble, store, or external to a controlspec.

2. The controlspec may contain any specific combination of control structures including decision table, state machine, petrinet, process activation tables.

3. The process activation table may specify control relations between the controlspec and the functional bubbles which are shown by control arcs but this is normally omitted.

4. The functional bubbles, actions of state machines, and transitions of petrinets may be made equivalent.

5. Triggers may cause transitions in state machines.

Notation:

1. Hatley/Pirbhai Notation

2. Ward/Mellor Notation

3. ESML Notation of Jensen et al.

Artifacts:

1. Control flow diagrams or Combination control and dataflow diagrams

2. Control specifications.

3. Decision tables (optional)

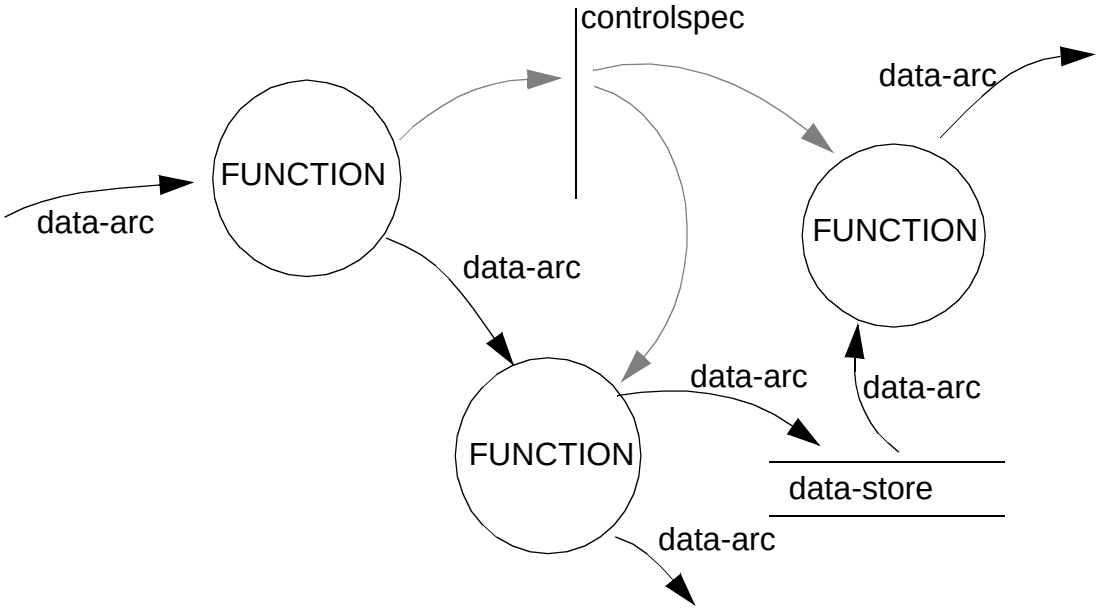
4. State machines (must be at least one or petrinet substituted)

5. Petri nets (optional)

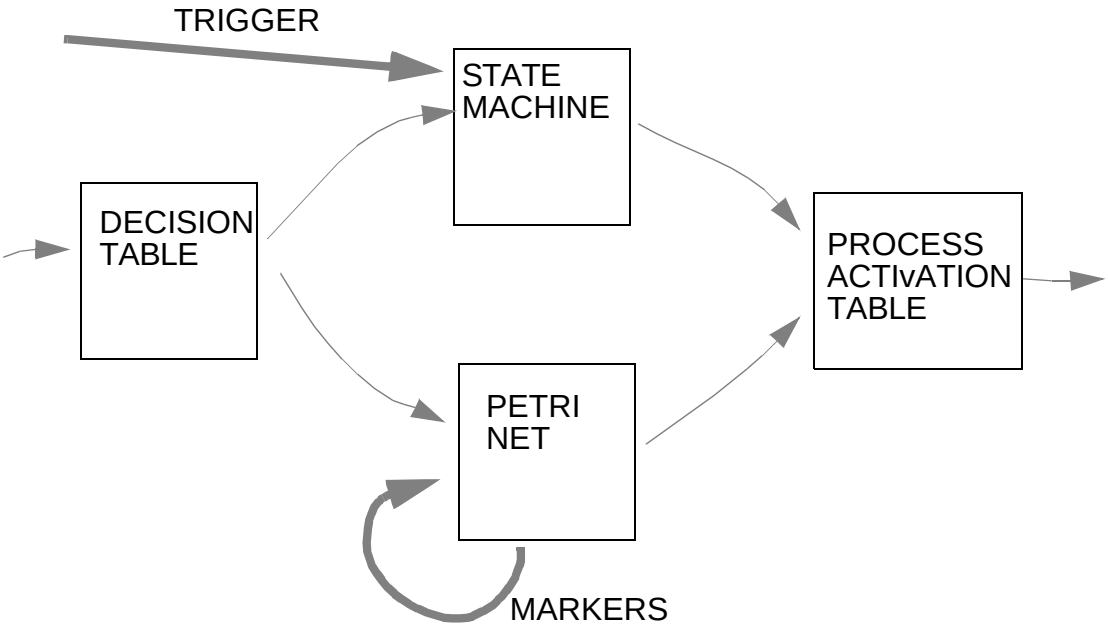
6. Process Activation Table (optional)

EXAMPLE:

1. Control flow diagram



2. Controlspec structure



ENTITY-RELATIONSHIP ARTICULATION METHOD

DESCRIPTION:

The Entity Relation technique allows us to specify any number of entities and their relations. Both entities and relations may have attributes. This allows us to see all the static elements in the system and their synchronic relations to each other.

CONCEPTS:

Tutorial:

Principles:

1. Entities are the persistent elements of a system.
2. Entities may have relations to each other.
3. Entities and relations may have types or belong to classes.
4. Entities and relations have attributes.
5. Entities represent the structural decomposition of the components of the system.
6. All entities should be encapsulated.
7. Entities are kinds and may have multiple instances.

Properties:

1. Entities may be decomposed into lower level entities.
2. Relations may be decomposed into sub-relations.
3. Attributes may be shared.
4. Classes have attributes shared by all the entities of a type.
5. Classes for an inheritance hierarchy.
6. The fundamental relations are IS_A signifying the belonging to a class and HAS_A which signifies ownership.
7. All the instances of an entity have the same general form.

Criteria:

1. The entity relation diagram should be as simple as possible.
2. The hierarchical nature of the entity relation diagram should be used to hide unnecessary detail.

Guidelines:

Measures:

1. Number of classes.
2. Number of entities
3. Average number of Attributes per entity plus maximum for all.

Notes:

1. See Chen for the origin of this technique.

ACTIONS:

1. Define Entities.
2. Define Relations between Entities
3. Sub-divide entities if necessary.
4. Define attributes for entities.
5. Group into Classes.
6. Define Class attributes.
7. Define Instances.

REPRESENTATION:

Elements:

1. Entity
2. Relation
3. Attribute
4. Classes
5. Instances

Relations:

1. Two entities are related.
2. Entities have attributes
3. Relations have attributes.
4. Classes have entities.
5. Classes have classes.

Notation:

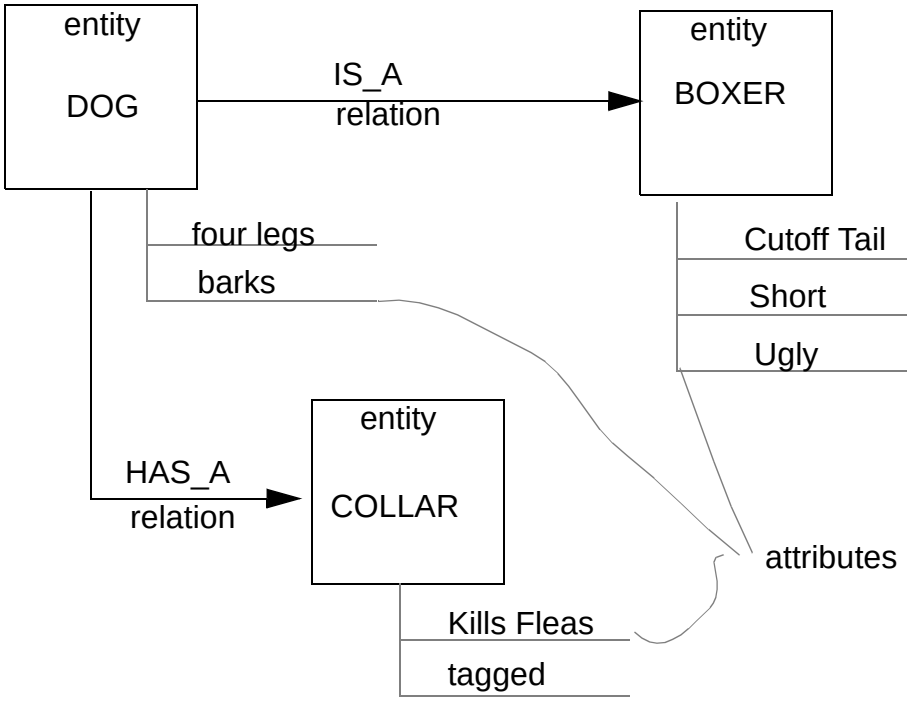
1. The inheritance hierarchy is show as a tree structure.
2. Entity relation diagrams have many notations. It normally appears as a network where the entities are nodes.

Artifacts:

1. Data Dictionary listing all entities and their attributes.
2. Entity-Relationship Diagram.
3. Inheritance Hierarchy

EXAMPLE:

1.



DARTS METHOD

DESCRIPTION:

DARTS means Design & Analysis of Real-time Systems. It is a methodology developed by Hassan Gomma which describes the agency aspects of embedded software systems. Basically the methodology identifies Tasks and the communication mechanisms which allow them to relate to each other and their operating environment. The methodology was taken up by the Software Productivity Consortium who developed ADARTS which is an elaboration of the DARTS methodology.

CONCEPTS:

Tutorial:

The actual architecture of a real-time embedded system must be expressed in terms of cooperating independent processing entities. At the lowest level these are tasks that share the resources of a processor. Tasks normally make it appear to the program that they contain as if it were running without obstruction whereas actually it is starting and stopping as it is allocated resources by some executive which controls the processor and its resources. Of course, there are units of processing below the task level called interrupt subroutines, but these are special cases which will not be considered here except to say that they may be considered special cases of the task scheme.

Tasks would left to their own run their programs without any reference to what other parts of the system is doing. In actual systems tasks normally need to coordinate their actions with other tasks and with the external environment of the platform and Local Processor Operating System (LPOS). The NGCR OSSWG Real-time System Reference Model gives a good picture of the normal environment for the development of embedded real-time systems defining all the pertinent interfaces. In the case of this method there are two types of important interfaces: between tasks and between tasks and shared resources.

There are many types of communications between tasks. These range from the Multiparty interaction as defined in MCC RADDLE & VERDI, to the Rendezvous in the Ada Programming Language, to the pipe that is used in UNIX, to the Queue which is normally used in real-time operating systems, to flags and semaphores.

If resources in a multi-tasking system are not controlled then deadlock and race conditions may occur. Therefore, Hoar developed what is called the monitor which is basically a control mechanism using a semaphore to allow tasks to fight over tokens and not the resources themselves. This solves most of the deadlock and race conditions that may occur.

Principles:

1. The system needs to be divided into a hierarchy of agents, normally called tasks.

2. Agents cooperate with each other through communication channels which carry messages.

3. Agents get their resources from controlled resource pools.

4. Distributed design needs to be distinguished from Tasking design.

Properties:

1. The tasking and monitor methods are analogous to the dataflow method where tasks are like func-

tion bubbles and monitored resources are like datastructures. Though there is an analogy it is exactly in the transformation of functions into task groupings that all the difficulty of designing real-time systems arises.

2. The methodology begins with the essential model of the functional decomposition of the system as a given.

3. Functions are grouped into tasks.

4. Tasks are connected by communications mechanisms.

5. Tasks are connected to their resources using monitors.

6. Distributed design deals with the aspects of system performance that accrue from distribution in space of the system.

7. Tasking design proper deals with the aspects of the design that have to do with time and the sharing of processing and other resources within a processor.

8. Tasks have ports that receive communications from a particular mechanism.

Criteria:

1. There is normally no one to one mapping between function-bubbles and tasks.Unless a system is very simple the cross mapping between functions and tasks will expected to be complex.

2. Tasks are constructed by taking lowest level functions and grouping them according to multiple criteria. This grouping may occur in many ways and different ways should be tried by giving the grouping criteria different criteria. In ADARTS these criteria are enumerated.

3. Tasking should be minimized in a system as context switch time may degrade performance. But where performance is not a consideration then tasks may be used to segregate groups of functions in a logical manner. Between these tow extremes there may be many ways to segregate functions into a task within any particular software system.

4. Good tasking first meets performance goals and secondly cuts down on the use of the communications channels.

5. Architectural design is done when all distributed and tasking design issues have been solved for the system.

Guidelines:

1. Do distributed design before tasking design.

2. Distributed design may occur on many levels depending on how processors are connected to each other. Deal with each level of distributed design separately.

3. Tasking design depends on the hierarchy of tasks. Do not use a tasking hierarchy as allowed in Ada unless absolutely necessary.

Measures:

1. Number of tasks.

2. Number and type of communication channels.
3. Number of levels in hierarchy of tasks.

Notes:

1. See Hassan Gomma on DARTS.
2. See SPC on ADARTS.
3. See RADDLE and VERDI from MCC.
4. See DURRA TASK DESCRIPTION LANGUAGE by Mario Barbacci of SEI/CMU.

ACTIONS:

1. Distributed Design
2. Define Tasks by grouping functions.
3. Define Tasking Hierarchy.
4. Define Communication Mechanisms.
5. Define Resource Monitors.
6. Define internal structure of tasks.

REPRESENTATION:

Elements:

1. Task
2. Ports
3. Communication Mechanism
4. Monitor
5. Distributed Communication Channel.

Relations:

1. Task to task via ports using communications mechanism.
2. Task to monitor using ports.

Notation:

1. Ready Systems had a task design tool that has a notation for showing the relations between tasks. It uses communications mechanisms and other features specific to their real-time operating system.

2. The DURRA Tasks Description Language has a formal language that describes tasks and their relations via ports. The statements in this language is used to generate source code for the tasking structure.

3. TRW NAS is a tool that allows tasking diagrams to be drawn and then these are used to generate simulations.

Artifacts:

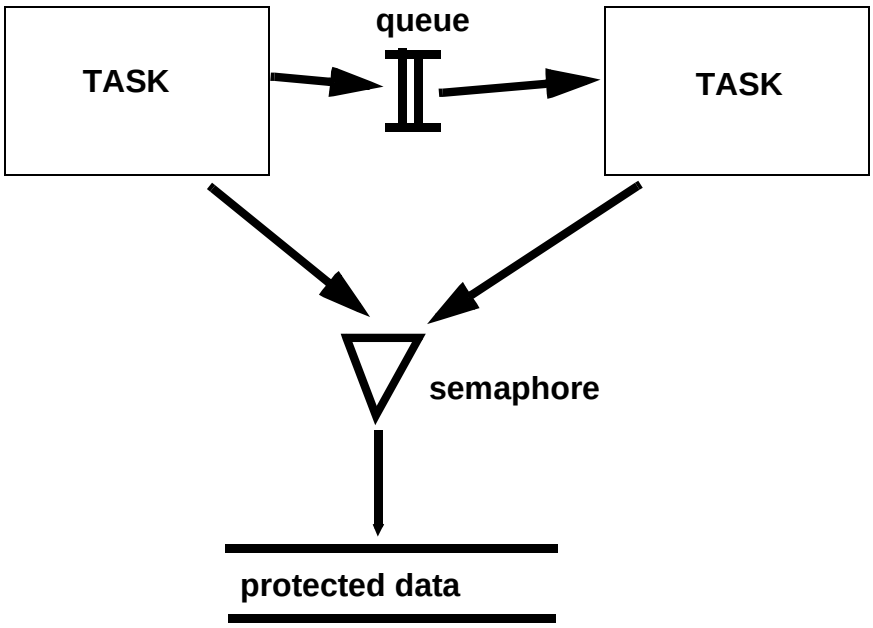
1. Distributed Design

2. Tasking Design

3. Global System Design

EXAMPLE:

1. Structure of tasking method.



ALLOCATION METHOD

DESCRIPTION:

The allocation method actually connects the functional and tasking views of the software design. It allows requirements traceability to occur throughout the design process thought the transformation of the essential model into an implementation model. This process is described in detail in the third volume of Mellor & Ward's Structured Real-time System Design.

CONCEPTS:

Tutorial:

The allocation of functionality to tasks and the trace back from tasks to functionality is a fundamental part of software design that is often not recognized as important. Allocation is a two way street which moves back and forth between tasking design and functional decomposition in order to continually assure the compliance of the design with the constraints of the requirements. This is done by starting design with an essential model and using the lowest level functions grouping them into tasks and then by modifying the essential model iteratively to embed implementation decisions within it.

Principles:

1. Must start design with an essential model.

2. Use the essential model to derive tasks.

3. Map back implementation decisions into the essential model transforming it into an implementation model.

4. The implementation model reflects the mapping of requirements to design at every stage of the design process.

Properties:

1. The essential model and implementation use the Hattley/Pribhai or Ward/Mellor methodologies.

2. The mapping is essentially a cross reference between tasks and functions.

Criteria:

1. The essential model should express functionality in an ideal environment of infinite computing speed, infinite memory, instantaneous communication, no distribution, and no tasking.

2. The implementation model should track the distributed and tasking design modifying the essential design.

Guidelines:

Measures:

1. Complexity of mapping.

Notes:

1. See Mellor & Ward Structured Design of Real-time Systems.

ACTIONS:

1. Bring Implementation model up to date.

REPRESENTATION:

Elements:

1. Same as Data & Control Flow Methods.

Relations:

1. Same as Data & Control Flow Methods.

Notation:

1. Same as Data & Control Flow Methods

Artifacts:

1. Implementation Model

EXAMPLE:

1. None

VIRTUAL MACHINE METHOD

DESCRIPTION:

The Virtual Layered Machine methodology was developed by Nielsen and Shumate in their book *Designing Large Real-time Systems in Ada*. It is an elaboration of the structure chart method which is a traditional way of describing software structure.

CONCEPTS:

Tutorial:

The basic concept of the Virtual Layered Machine method is that for any given problem it may be solved at some level of abstraction by a set of instructions which are used to solve that particular design problem. These instructions form a virtual machine and any particular instruction may in turn be considered such a machine until the problem is completely decomposed and taken down to its most concrete level of description. At any level Instructions may be composed of operations on objects, pure transformations of non-persistent data, or sub-instructions.

Principles:

1. Every solution to a software design problem may be expressed as set of instructions which work together to solve the problem.
2. Instructions may be decomposed into sub-instructions.
3. Sets of instructions may operate as an Automata or Petrinet.
4. Lowest level instructions are operations of the persistent data of objects in most cases.

Properties:

1. Instructions may contain instructions, operations, or functions.
2. Sets of instructions form a whole called a Virtual Machine.
3. Machines may be associated with Automata or Petrinets.
4. The virtual machines form a hierarchy.

Criteria:

1. Any

Guidelines:

1. One may build virtual machines from bottom up or top down or any combination that makes sense.
2. Any level of abstraction may be picked as the starting point.
3. The virtual machines mapping to the structural decomposition hierarchy of the essential model should be maintained.

Measures:

1. Number of levels of virtual machines.
2. Number of instructions.
3. Number of Objects and Operations.
4. Number of Automata or Petrinets.

Notes:

1. See Nielsen and Shumate's *Designing Large Real-time Systems in Ada*

ACTIONS:

1. Select level of abstraction.
2. Produce instructions set to solve design problem.
3. Decompose Instructions to get next lowest level.
4. If you know about Objects and their operations design them as you go along.
5. Attach Automata or Petrinets as appropriate.

REPRESENTATION:

Elements:

1. Instruction
2. InstructionSet
3. VirtualMachine
4. Object
5. Operation
6. Action
7. Transit

Relations:

1. Instructions to Instruction
2. Instruction to VirtualMachine
3. VirtualMacine to Automata or Petrinet
4. Instruction to Object Operation

5. Instruction to Automata Action
6. Instruction to Petrinet Transit

Notation:

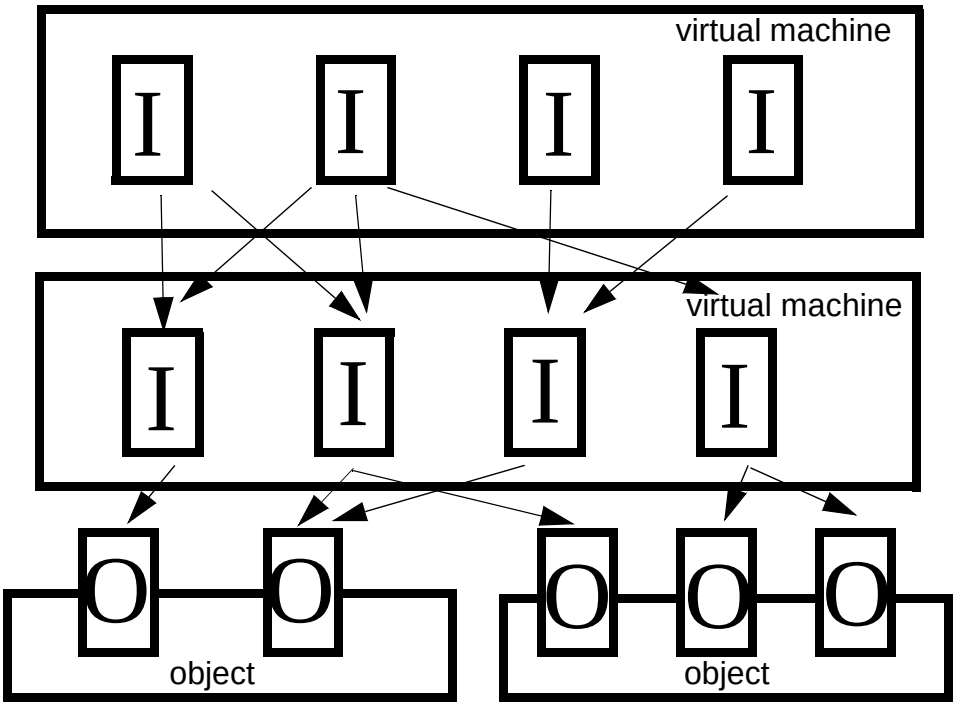
1. Structure Charts may be used.
2. Nielsen & Shumate suggest a graphical notation.
3. Tony Wasserman of IDE has suggested a generic Object Notation that has many of the necessary features.

Artifacts:

1. Structure Chart.

EXAMPLE:

1. Structure of a hierarchy of Virtual Machines.



WORLDLINE & SCENARIO METHOD

DESCRIPTION:

This pair of minimal methods allow the interaction of agents to be traced and interrelated through time even though each is traveling its own world line the agents inteact in scenarios which allow causal waves to be determined.

CONCEPTS:

Tutorial:

This pair of methods invokes Special Relativity which appears in distributed systems as the impossibility of a reliable global clock. Each agent appears to have its own independent timeline called a worldline which represents its view of everything that is happening inside and outside the system. Each agent has its own point of view and things may happen at a very different time than they appear to happen from the point of view of another agent. What we are really interested in developing here is how the events of the rest of the system and environment appear from the point of view of every agent and also what are the possible chains of causality which link these appearances together.

Principles:

1. Every agent has their own clock.

2. Clocks may be set to reference clocks but that takes time.

3. There is the possibility of time warp where an individual agent can go ahead and make calculations given what appears to it as good data as long as it maintains a history and can reset if it finds out some of its data is wrong.

4. The wave of finalized causality reaches each agent at different times and it is not possible to determine when that is exactly. It is basically when no more corrections to a message arrive.

Properties:

1. Agents have their own time line.

2. Communication channels connect timelines.

3. Causality travels via communications channels.

4. Agents may interact with groups of other agents in a single INTEACTION.

5. Worldlines are generated from role templates.

Criteria:

1.

Guidelines:

1.

Measures:

1. Number of Agents.
2. Number of Messages
3. Number of roll-backs

Notes:

1. See Time-Warp Simulation systems.
2. See Einstein's explanation of Special Relativity.
3. See MCC RADDLE and VERDI

ACTIONS:

1. Trace message paths.
2. Trace actions of each agent through time.
3. Produce Roles that generate timelines of each Agent.

REPRESENTATION:

Elements:

1. Agent.
2. Timeline
3. Scenario
4. Role
5. Message

Relations:

1. Agents receive Messages
2. Agent has sequence of Events.
3. Scenario contains Events across worldlines.

Notation:

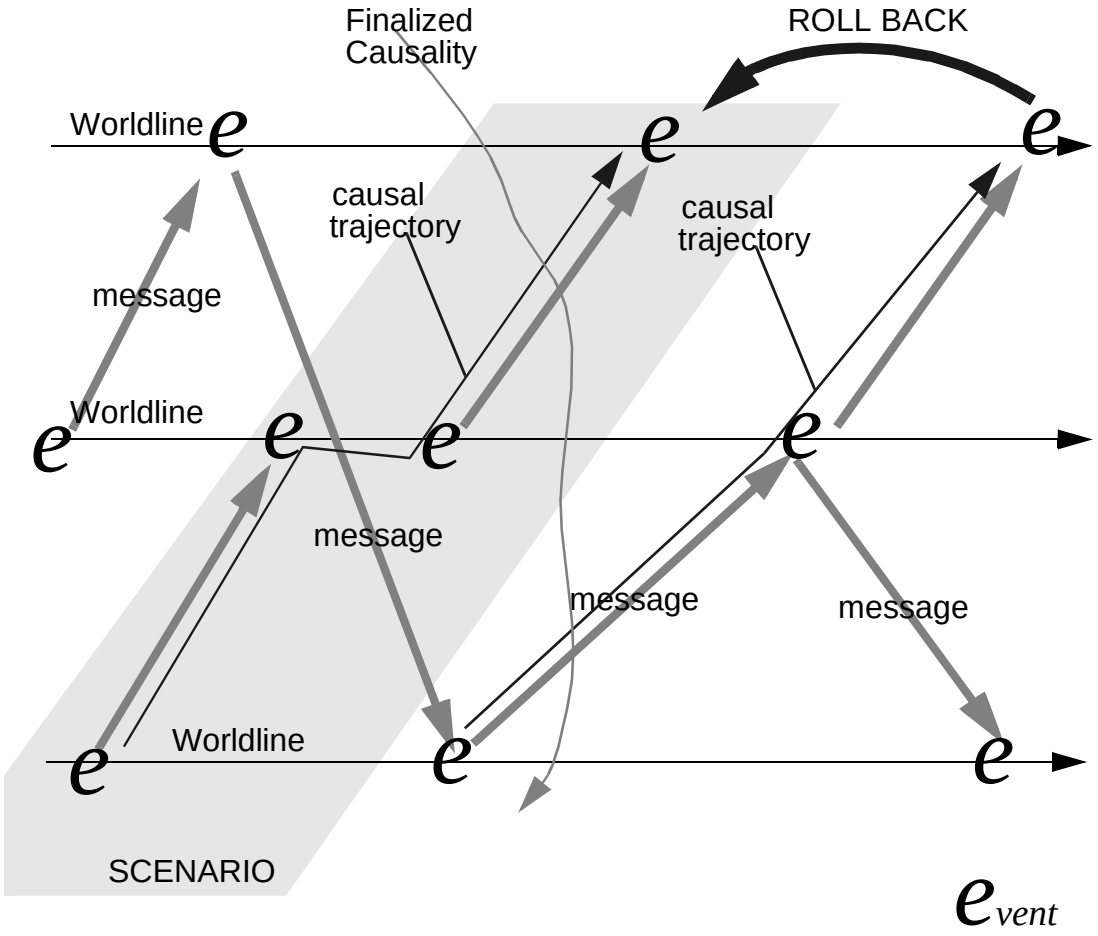
4. See MCC RADDLE & VERDI
5. Agha ACTORS

Artifacts:

1. Scenario descriptions
2. Role diagrams
3. Timing diagrams

EXAMPLE:

1.



DESIGN ELEMENT AND INFORMATION FLOW METHODS

DESCRIPTION:

The fundamental methods related to spacetime gage the flow of information and design elements through the system. Information flows between variables and can be looked at in terms of all the data that are in all the variables at one time (synchronically) or by following one or more pieces of data as they flow through the system and are transformed (diachronically). Design Elements also flow through the software systems whose structural elements are malleable and even self modifying. Design elements states may be compared to system states and design element actions may be compared to system actions.

CONCEPTS:

Tutorial:

Information systems may be looked at either as state machines or information networks. As state machines they take inputs and produce outputs so one may look at the discrete state of variables at successive points in time. As information networks it is possible to instead look at the way data flows through the network and is transformed in the process. From the information network point of view the information is malleable whereas from the state machine it is the states of the variables that are malleable.

Information systems have data that flows through them and also data that compose them. The compositional data are called design elements. Design elements are all little state machines that must relate to the overall state machine of the system as a whole or some higher level aggregate. Design elements flow through the system in the same way information does except at each state there is a correspondence maintained between overall states and lower level design element states. This correspondence may either be looked at by looking at systems states and design element transitions or system transitions and design element states.

Principles:

1. Every software system has a set of variables.

2. You can look at the contents of all the variables at once or a single piece of information flowing through several variables.

3. You can look at variables as the state variables of automata.

4. You can look at the coordination between overall system states and design element states

5. Either you look at system states and design element transitions or design element states and system transitions.

Properties:

1. Variables have values.

2. Sets of values in all system variables at one time is the aggregate system state.

3. Information flows through variables represented as values.

4. Variables may be considered to be individual automata.

Criteria:

1. Overall the goal is to make the design element states coherent as possible with the overall system states.

2. Only necessary information should be kept in the system variables at any one item.

Guidelines:

1. Information flow and information network are the basic debugging information for software systems.

2. All variables should be treated as state machines and explicitly coordinated with overall system states.

Measures:

1. Number of Variables

2. Number of Information flows.

3. Number of Design Elements

4. Number of states and transitions per design element

5. Number of System states and transitions.

Notes:

1. See David Gelernter Programming Linguistics for an excellent generalized model of these space-time relations as an abstract programming language model.

ACTIONS:

1. Design information variables for design elements.

2. Describe information network that connects variables.

3. Describe information flows through network

4. Describe coordination of overall system states and design elements states.

REPRESENTATION:

Elements:

1. Variable

2. State

Relations:

1. Relation between Variables.
2. Sets of variable relations represent Information network.
3. Information flowing through network through channels
4. Relation between states which are values of variables
5. Actions of Automata or Petrinets cause changes in values.

Notation:

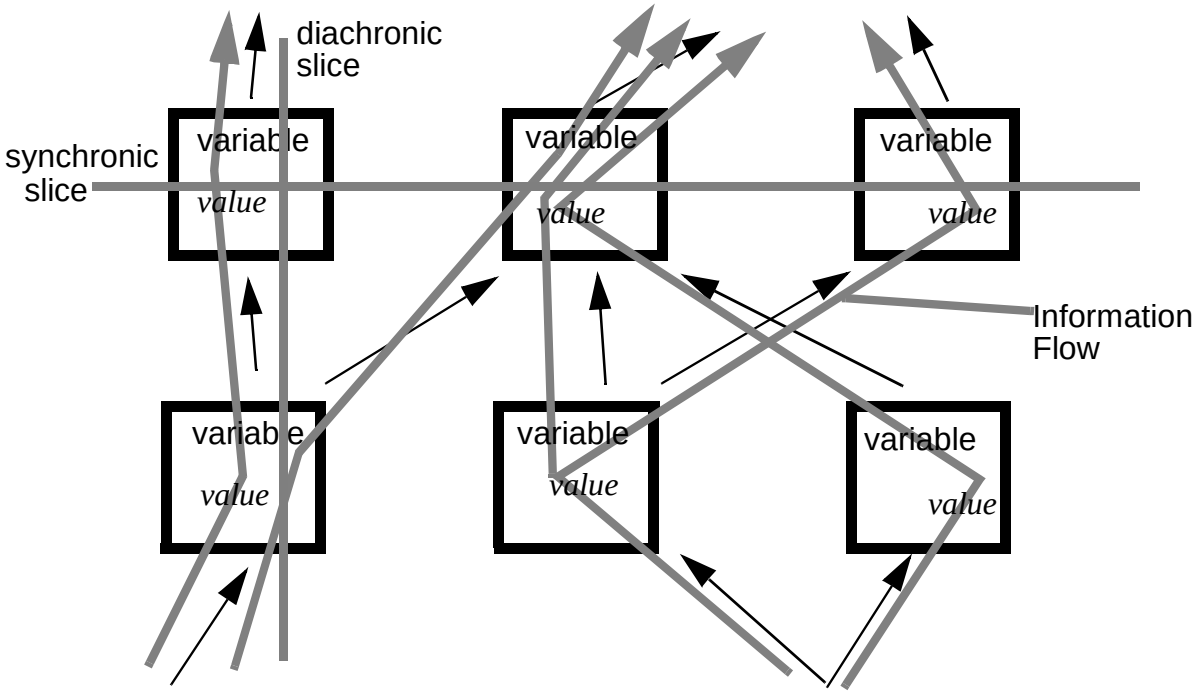
1. Gelernter PROGRAMMING LINGUISTICS
2. SOFTWARE DIAGRAMMING

Artifacts:

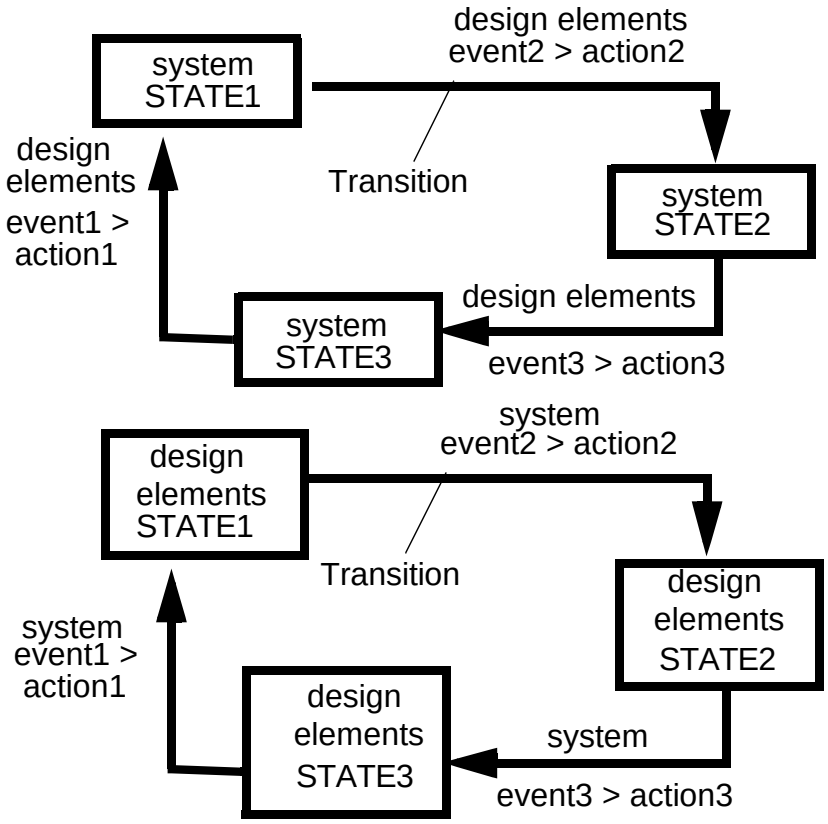
1. Information flow diagram
2. Information network
3. List of Variables in System
4. Design element states and actions
5. System states and actions

EXAMPLE:

1. Information network and information flows



2. Design Element Flows in system context



TEMPORALITY METHOD

DESCRIPTION:

In every system time must be described. It is normally described in terms of events and duration.

CONCEPTS:

Tutorial:

Allen has provided us with an interval logic for the description of time through constraints on the ordering of events.

Principles:

1. Events are fully ordered (linear with distance) in a software system.
2. Constraints operating between events may be described using a temporal logic.
3. Events are fluctuations in signals.

Properties:

1. Events may have the following relations:
    - Before
    - After
    - During
    - Starting
    - Finishes
    - Overlaps
    - Meets
    - Equals
2. SIgnals come in bundles which in turn form sheaves.
3. A single signal by itself is meaningless.
4. Bundles split and join describing branching points in possible system timelines.
5. The fundamental signal is the clock pulse.

Criteria:

1. All temporal constraints should be satisfied by the system.

Guidelines:

1. Timing diagrams should be as simple as possible with only the most crucial signals portrayed.

Measures:

1. Number of Events

Notes:

1. See Temporal logic for relations of causality and necessity in sheaves and between bundles.
2. See Allen's Interval Logic

ACTIONS:

1. Define events.
2. Produce timing diagram.
3. Work out constraints between events using Interval Logic
4. Work out necessities using Temporal Logic.

REPRESENTATION:

Elements:

1. Signal
2. Bundle
3. Sheaf
4. Event
5. Interval
6. Duration

Relations:

1. Signals contain Bundles
2. Bundles contain signals
3. Signals have events as fluctuations.
4. Gaps can only be measured by comparing signals.

Notation:

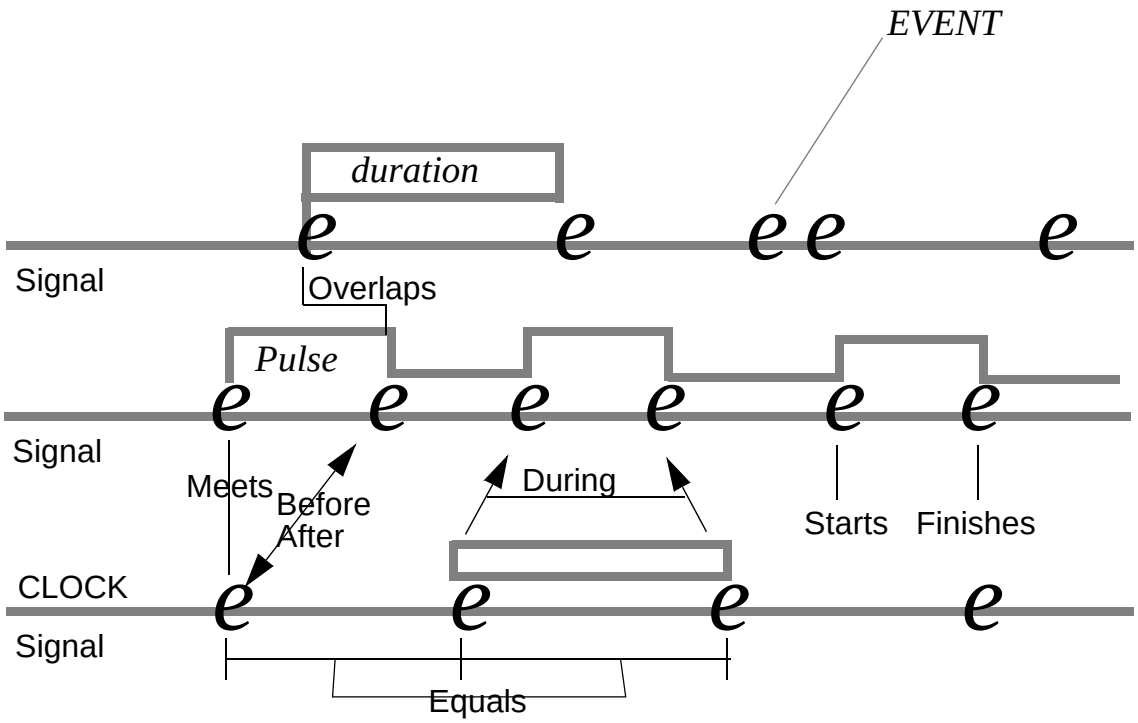
1. Allen's temporal logic.

Artifacts:

1. Timing Diagram

EXAMPLE:

1.



Apeiron Press

PO Box 4402
Garden Grove,
California 92842-4402

714-638-7376
714-638-1210
palmer@think.net
palmer@netcom.com
palmer@exo.com
Dataline 714-638-0876

This book was set using Framemaker document publishing software by the author.

Electronic Version in Adobe Acrobat PDF available at http://server.snni.com:80/~palmer/homepage.html

Library of Congress
Cataloging in Publication Data

Palmer, Kent Duane

WILD SOFTWARE META-SYSTEMS

Bibliography
Includes Index

1. Philosophy-- Ontology
2. Software Engineering
3. Software Design Methods

I. Title

[XXX000.X00 199x]
9x-xxxxx
ISBN 0-xxx-xxxxx-x

Keywords:

Software, Design Methods, Ontology,
Integral Software Engineering
Methodology, Systems Theory