SOFTWARE ENGINEERING FOUNDATIONS

A Paradigm for Understanding Software Design Methods

Integral Software Engineering Methodology

Kent D. Palmer, Ph.D.
Software Engineering Technologist
PO Box 4402 Garden Grove CA 92642
palmer@netcom.com

The Integral Software Engineering Methodology (ISEM) has grown out of our previous studies of the software engineering foundations. The main purpose of this paper is to review the main aspects of a formal language for expressing software designs. That language has many unique features but is based on a set of empirically derived methodologies in current use by advanced software engineering practitioners. The language 'ISEM' will be used as an example to show what is necessary for a complete statement of software methodologies for the design of real-time software systems. The language is an experimental formalization of the knowledge structure that software systems engineers must have to design a real-time system abstractly. ISEM may serve as an example for the development of other future languages to fill the void between current partial design methodologies and what the practitioner

actually needs.

1.SUMMARY OF THE FOUNDATIONS

The basic lesson to be taken from the previous studies is that there are FOUR viewpoints on Software Design. These are the AGENT (who), FUNCTION (what), DATA (where), and EVENT (when) perspectives. Designs are themselves ultimately unrepresentable as Peter Naur suggests because the reasons why a particular design is the way it is are virtually infinite. Thus all we can expect are partial representations from a variety of viewpoints. A particular design element visible from one perspective may vanish or be transformed as you move to another viewpoint on the design.

Given the viewpoints methods show up as ‘bridges’ between viewpoints. Each viewpoint has its own method for representing the system independent of all other viewpoints, and also each viewpoint has a connection to methods which represent one of the other viewpoints from its own perspective. Thus between each viewpoint on software design there is either a two way bridge or two one way bridging methods that allow transitions between

viewpoints on the design. An example of a two way bridge is the Dataflow method. It allows easy transition between the Function and Data viewpoints going in each direction using the same technique. On the other hand between Function and Event are the State Machine and Petrinet methods which are duals of each other and each give a one way transition between these viewpoints.

The four perspectives plus the sixteen minimal methods (4 for the viewpoints independently and 12 bridges between viewpoints) give an overview of the field of all necessary methods on software design. These methods may be embroidered in many ways to allow refinements of design description. But, it is necessary to restrict the methods to their minimal complexity in order to get a view of the entire field of software methods which may be easily understood and applied. The ISEM language attempts to allow this minimal, or near minimal, representation to be captured so that the methods may be studied and their interrelationships to be made explicit.

The four perspectives are not equal. In the process of connecting them with the General

Systems Theory of George Klir it was discovered that the Agent and Function viewpoints are only partially ordered whereas the Data and Event viewpoints are fully ordered. This explains the differential complexity of the method bridges associated closely with these viewpoints. Minimal methods themselves are all structually based on the dual orders (Linear with no distance & Partial with distance). But in real-time systems an unbridgable gap appears between these two types of perspectives which can only be crossed empirically based on intuition. Thus there is some support for the fact that software engineering will always have some aspect as an art no matter how strong its scientific rendering. Because of this gap it becomes crucial to map the transformations of design elements between different perspectives on the design. This is the motivation of the creation of the ISEM formal design language.

The minimal methods on which ISEM is based is represented by an entity-relation diagram of core concepts which appears as an appendix to this paper. There is an important difference between ISEM and this conceptual core system. The conceptual core is meant to

represent the minimal methods necessary for software design adhering to the principle of Ocham's Razor. ISEM embroiders this by adding what are strictly unnecessary concepts to make things easier for the practitioner or to express a certain approach to the design of real time systems. The conceptual core of the language may be seen in the glossary by looking at the starred items as opposed to the capitalized which represent ISEM concepts. Where the conceptual core is honed down to the absolute minimal set of concepts necessary to represent all the minimal methods; ISEM on the other hand might be expanded to what ever is useful in the actual design process.

2. FORMAL DESIGN LANGUAGES

Programming languages are one well worked out kind of formal language which allows software implementation. However, as we move up the ladder of abstraction to the level where non-localization no longer appears as it does within the programming language, where all the design element's descriptions maybe localized, we find that there is a possibility of defining another kind of formalized design language at this level. At the present abstract

designs are normally represented by graphical notations in various methodologies. However, these methodologies and their associated enabling graphical tools do not cover all the necessary aspects of design so designers find them less than completely useful. In fact much of design still occurs on the back of envelopes, so to speak. ISEM seeks to create a formal design language at the correct level of abstraction for software designers work which also covers all the necessary aspects of design representation. This language could be rendered into a graphical notation when it has stabilized. However, the first point is to get a complete set of representations, instead of rushing into graphical notations which do not do the job. Also, the formal design language itself could be used to represent designs, in a form like source code, which many software engineers would prefer. All design languages need to offer both linguistic and graphical forms for software design because there are generally different types of people who are more accustom to one rather than the other. I have offered graphical forms taken from the literature for each of the core methods in the second part of this series. However, graphical descriptions lack precision and have a

limitation on the complexity they may describe. So even though I am a graphically oriented person, I was driven to create a formal design language in order to cover express everything that needed to be expressed in software designs.

ISEM is not like VDM or Z. Each of these formal languages are generic expressions of the principles of logic and abstract mathematics which may be applied to describe and constrain software specifications. They lack an explicit model of software architecture. ISEM attempts to give that architectural model an so VDM or Z are actually complementary to ISEM and may be used to constrain its architectural designs in the same way they now constrain specifications.

3. THE DESIGN OF ISEM

ISEM was designed with some very different goals from those that normally drive the development of programming languages. First ISEM was not meant to support implementation at all. So it lacks all the constructs that normally appear in a programming language. Instead ISEM would

basically allow the definition of design elements and their connection to each other into an architecture. Second, ISEM wants to support incremental definition. So that if the designer know any one fact about the design he can record that without having to know everything about a particular design element. This means if the designer knows about one state of a state machine he should be able to state that the state machine has that state and nothing else. In an implementation language one normally has to know everything about a state machine in order to write its implementation. In design one wants the design facts to build up one by one until one knows enough to restate it in a more concise form. Third, ISEM was to be a language with no deep structure. Thus it should be composed of single statements like those of the BASIC programming language. The recombination of language components using the deep structure of the language would not be allowed. The ISEM language has only surface structure. This simplifies the language and allows it to be used to not just state designs but to analyze design relationships. Each sentence in ISEM is an explicit design relationship. Fourth, the ISEM language is open ended. It allows new

statements to be added by the designer as needed. So the language dealt with in this paper is only the kernel of a language that would be extensively developed and modified in use.

These are some of the ground rules in the development of the ISEM language that makes it very different from a normal programming language. The fact that specialized formal languages can be constructed has not become widely very widely known. People usually think of creating their own programming language but rarely do they think about creating their own formal non-programming language to describe some phenomena. One exception is the biologists who have begun using formal languages to describe biological phenomena. The point is that once a formal description exists using a specialized formal language then it may be accessed as a database or it may be used to drive a simulation even though it cannot be compiled. The descriptive force of formal languages is very great since a few language production rules may express a great range of concrete design possibilities. For instance the AUTOMATA sub-language can express all possible state machines and it only has twenty or thirty statements. Thus the great

boon of formal languages is their explicitness and their fund of expressiveness.

The ISEM language is not meant to be complied instead it is meant to serve as a database of the current design while it is being formed and changed. It may also ultimately serve as the basis for simulation of the design in order to prove certain properties before the design is implemented. In this mode it will serve as a means to explore the design space. And this is the most important function that the ISEM language could fill. Now there is very little exploration of the design space of systems. This is because designs are so complex and they cannot be simulated. However, once we have a complete formalization for design representations it will be possible to simulate many static and dynamic aspects of the design and also change that design so as to find out what other design configurations have to offer for improved satisfaction of crucial required parameters.

So the ultimate aim of ISEM is toward design simulation. However, until we have simulators to support formal design languages ISEM can serve to represent concrete designs and to study

the nature of design representations themselves. We need to know what are the minimal set of design statements needed to represent all the aspects of real time systems. Then we need to know what are the set of extensions that designers most often use. These two sets of statements would make up the ISEM language. Then each designer would add their own extensions which reflect their applications area or their own understandings of the correct elements necessary for their designs. But ISEM has another function. It has an educational function in the movement of our craft from the beginnings of the commercial stage into the stage of professional engineering. The ISEM language represents a kernel of design knowledge and a way of organizing that knowledge that new or untutored experienced software engineers can learn from. ISEM represents the formalism of design concepts which every software engineer should be familiar with and be able to apply. It is the foundation on which design heuristics can be built which would allow the neonate to learn from the more experienced engineer. It also contains a conceptual structure that will serve as the a solid center for the learning software engineer to attach the knowledge as he learns

how to design different applications. But more than that it is a language that will allow software engineers to express their designs for mutual elucidation and criticism so that using this formalism designs should be more accessible, visible and understandable.

Design formalisms are still in their infancy. This one has its own peculiarities given its origin in concerns about the philosophical and system theoretic foundations of software practice. It is however an example upon which others may build or transform for their own purposes. It is the beginning of a foundation for software engineering as a professional discipline because it makes the basis of software engineering knowledge explicit. What software engineering has to add to Computer Science beyond the concern with the sociological and managerial dimensions of the actual production of software is the knowledge of software design methods that allow systems to be designed at higher levels of abstraction above the level of source code. Software methods are the kernel of this new discipline and ISEM attempts to survey the whole field of necessary methods based on the organizing framework of the four perspectives and the idea

that there are 16 minimal methods necessary to represent all aspects of real-time software design.

4. THE STRUCTURE OF ISEM

ISEM doesn't really have a grammar like we are used to with programming languages. It was consciously designed to be like the BASIC programming language which only has a series of statements with minimal substitution. This will allow the language to be built up incrementally because separate statements may be added without recompiling the parser like a macro language which are used to extend some applications. Instead multiple levels of substitution ISEM has a template for the formation of design statements. This pattern is used to form all ISEM statements and represents a configuration of basic parts of speech.

STATMENT: ***AC1***

Short Title: ***PositWrkrDoesActInStateOnEvent***

Long Title: ***PositWorkerDoesActionInStateOnEvent***

SUB-LANGUAGE: ***ACTOR:*** *{which grammar is optional}*

OPERATION: ***POSIT:*** *{statement operator, first noun}*

NOUN: ***WORKER*** *{statement operand}*

Identifier: ***worker_name*** *{instance identifier}*

VERB: ***DOES*** *{statement verb}*

PREPOSITION: ***none*** *{first connector}*

NOUN: ***ACTION*** *{second noun}*

Identifier: ***action_name*** *{instance identifier}*

PREPOSITION: ***IN*** *{second connector}*

NOUN: ***STATE*** *{third noun}*

Identifier: ***state_name*** *{instance identifier}*

PREPOSITION: ***ON*** *{third connector}*

NOUN: ***EVENT*** *{fourth noun}*

Identifier: ***event_name*** *{instance identifier}*

PREPOSITION: none {fourth connector}

NOUN: ***none*** *{fifth noun}*

Identifier: none {instance identifier}

Qaulifier: ***none*** *{attribute}*

Yindex {the current sort position of all ISEM statements}

Zindex: {sequential statement number, does not change}

The template allows up to five nouns, basic ISEM entities, to be connected to each other in

two different structures. The first structure is an Operator--Operand structure where the operation works on the Operand which is then modified by the other nouns. The second structure follows the SVO of English and allows a Subject to be specified by the first noun followed by a verb and then four objects. Objects are related to the first using prepositions. All the parts of speech in the template are slots that may be empty or full depending on what the designer is attempting to express. So in this case the fifth noun and it's preposition positions are left empty. The Operator slot is filled with the token 'POSIT:' which is optional. Optional words have a colon appended to show that they are not necessary to the sentence. Generally all the action statements are will use the operator-operand form while all the descriptive statements which describe the structure of the design will use the SVO sub-pattern.

EXAMPLE:

WORKER onEvent1 DOES ACTION setClock IN STATE waiting ON EVENT reset.

In a design the pattern will have it's instance variables filled in as in the example above.

Notice that every identifier follows its type so that it is always clear what type any particular token is as opposed to most programming language where you have to remember what type a particular token is. This allows the sentence to function as a database in which the type of a field precedes the contents of a field. One may always search for all ‘ACTIONS’ or all ‘EVENTS’ or what ever one needs to find using standard queries directly on the design text itself. Or the design may be entered into a database set up on this pattern and the database could be queried. The pattern is simple so the designer can make up his own statements with relative ease. The basis idea of the template is to allow a minimal system of design elements to be operated on at one time and connected to any other design element. It turns out that the language as it stands does not use the fifth noun or the fourth preposition. These places are reserved for more complex situations not covered by the language as it stands.

The language itself is divided into sub-languages. Each sub language addresses a particular minimal method or some associated realm of description necessary to represent designs. The following languages have been

incorporated to date:

ACTOR: Describes the parallel agents perspective based on Gomaa's DARTS and Agha's Actors.

ARCHITECTURE: Describes the generic software architectural elements.

AUTOMATA: Describes State Machines.

DOMAIN: Describes the application domain and requirements analysis extensions to the methods.

INFORMATION: Describes information network within the system.

LIST: An augmented list.

MACHINE: Describes a virtual layered machine as formulated initially by Neilson & Shumate.

PROCESS: Describes dataflow and control flow techniques as defined by Hatley/Pirbhai.

PETRINET: Describes Petri News formalism.

SITUATION: Describes the situational aspects of the active objects design which involves placement of data.

SET: An augmented mathematical set.

SYSTEM: Describes the software system as a whole in relation to the environment and meta-systems.

TEMPORALITY: Describes the time aspect of the system through temporal logic.

TURING: Adds systems mechanization elements that make it possible to unify the design description.

Named after Alan Turing.

WORLD: Describes the worldline and scenario minimal methods that show how different actors communicate within the relativistic world.

There are currently over seven hundred statements in all the sub-languages taken together. ISEM is merely the sum of its sub-languages and it may be that various other sub-languages are added as their necessity becomes apparent. So ISEM is extensible both within a sub-language and also by adding other sub-languages. For instance a logical sub-language could be added to fill the place of VDM or Z; or those formal languages could be used in conjunction with the ISEM sub-languages. The guiding concept behind ISEM is flexibility. Since ISEM is not tied to a compiler ti my grow and change easily.

5. THE ISEM SET AND LIST SUB-LANGUAGES

As an example how the ISEM language is constructed we shall first look at the SET sub-language. The SET sub-language has all the necessary statements to construct any set. For instance:

(01) DEFINE: IDENTIFIER bag IS SET.

(02) BAG SET bag1.

With this statement (01) a SET is created and with the next (02) it is turned into a BAG in which different instance of the same kind may be stored. Here ‘bag1’ is the identifier of an instance of type SET which we now know is a BAG.

(03) POSIT: ELEMENT x, 1,2,2 BELONGS TO SET bag1.

Wherever there may be an identifier there may be a list of identifier unless there is a special rule produced to exclude it. So here the elements “x, 1, 2, 2” are placed in “bag1”. Now we know why is in the bag. If ISEM were automated we could query what we had put in the bag.

(04) INQUIRE MEMBERSHIP OF SET bag1.

And presumably the design system would respond something like. . .

bag1 = x, 1, 2, 2.

So it is with all the aspects of the ISEM language. In fact, all this seems so simple the question at once appears why hasn’t anything

like this been done before this. The answer is that there have been many partial attempts to describe designs and some automated tools have been designed to enable these descriptive techniques. ISEM really only brings the idea of using a human readable formal design language to describing all of the pertinent aspects of design. So what is described here is well within our reach technologically. In fact as an industry we have gone to lengths to develop means of depicting designs graphically when we had the tools to develop formal languages already. Since we skipped over the task of developing the formal language that fully expresses the design it is necessary to now go back and do that. The form tools we now need are both graphical and linguistic. An example is the languages that some document production systems have for describing text documents with graphics (for instance, Maker Interchange Format of the Frame Maker document processor available from Frame Technologies). You can draw a picture in Frame Maker then look at the MIF which is more or less human readable. You can change the MIF and see the modified picture when you read the document back into the document processor. The ability to move back and forth between graphical and

textual representations is crucial to our ability to do design work. When we are doing some construction graphics is appropriate while for other construction work text is more appropriate. With current systems what we lack is the formal language to underlie the graphics. In some ways we have the cart before the horse. We solved the exciting technological problem and left unsolved the hard problem of what should be in a design formalism, what are all the crucial aspects of real-time systems at the level of methodological abstraction. It is really only this hard unanswered question that ISEM takes a crack at. It is the first in what will undoubtedly be a series of formal languages to express the essentials of real-time design. Once we have that, and it is more or less agreed on then software engineering as a discipline is going to take a giant step toward becoming a professional engineering discipline. Of the vendors, to my knowledge, it is Interactive Development Environments's toolset called 'Software Through Pictures' that comes closest to this vision. Underlying each of their editors is a data language which mediated between the graphical representation and the underlying database. The real problem they have not addressed yet is that this language must be

easily human readable. That the designer should be able to change it directly and see the new pictures, and that all the sub-languages need to interact properly. Now IDE's editors are all separate languages and graphical notations with minimal mostly behind the scenes connection as dictated by methodologies. Instead I envision a similar tool set on top of a unified design language.

IDE has gone the crucial step of recognizing that formal languages have an important place in their system. Now what I am interested in is starting the process of defining that language.

The ISEM SET sub-language is an example which all the other sub-languages follow the broad outlines of in the way they work. If you look at the SET entity relationship diagram or the list of SET language statements you will see that there are some statements that construct the essential relations and another set of statements that do operations on those relations. The basis steps are always DEFINE the instances; POSIT the relations between instances based on design element types; produce LEMMAs in which specific temporary attributes or relations are set up; and

OPERATE on the instances and there configurations. You can also see that there is a statement which will transform a SET into a LIST. Other auxiliary operations will transform design elements in various ways.

There are many ways that a SET language might be designed. This is only one of a myriad styles of formalism. The style is inessential. The essential point is that we need to learn to use these formal languages and that software engineering has a linguistic aspect that goes beyond what logical and mathematical formalisms can offer. This linguistic aspect of software engineering methods is crucial to the development of the discipline. It is introduced with the partially ordered AGENT and FUNCTIONAL perspectives. What is lost with the inability to fully order agency and functional aspects of a system is gained back with the ability to express meaningful relationships between design elements.

In what follows we will take a tour of all the different languages and mention the peculiar features and generally attempt to point out how the structure of the design elements described by the sub-languages fit together. For the most

part the sub-languages are self explanatory. If you see something that is missing remember this is just an example and the language is extensible to add relations that I did not think about when making up the individual statements. What I have attempted to do is provide a core set of design constructs that covers all sixteen minimal methods and some other concepts which I deemed necessary to make the language workable for the designer such as the SYSTEM, ARCHITECTURE, and DOMAIN sub-languages.

I will follow the order of the ISEM entity relationship charts. These charts are provided so the reader can get a quick overview of the languages and the relations posited by each statement. The statements often work together and have specific interrelations which show up well in these diagrams. It is difficult for us to read the language directly and see these relationships. But what the language lacks in graphical intuitiveness it make up for by its formal explicitness.

A glossary is provided to give some explanation of the individual design elements. Capitalized words in the following guided tour

indicated design elements that may be found in the glossary.

6. SYSTEM

Notice that many times a generic design element like SYSTEM will have three associated statements:

DEFINE: IDENTIFIER id IS SYSTEM.

DEFINE: IDENTIFIER id IS SYSTEM OF TYPE id.

DEFINE: SYSTEM id IS TYPE id.

These allow a SYSTEM to be identified without specifying its type. Or one may specify the system and its type together. Or one may take a system identifier that already exists and specify it's type. This shows how the language allows incremental build up of determinations of the design. This aspect of the language is very important as design is an iterative and recursive non-routine enterprise.

The system sub-language describes the relationship between the SYSTEM and it's ENVIRONMENT and any METASYSTEM that it operates in relation to. Generally current methodologies are poor at describing these

relations which are often as crucial as the internal articulation of the system. systems, metasystems, and environments all have hierarchical decompositions, possibly multiple ones which conflict. Being able to explicitly define any HIERARCHY is of use in defining the envelope of the system and how it interacts with the outside world. Context diagrams are often too simplistic. Since systems, metasystems, and environments are all multiply hierarchical and since interaction may occur between any levels in any hierarchy the context diagram will not give us the necessary depth of representation needed for complex real-time systems. This sub-language allows us to describe any and all such situations explicitly. Each system and metasystem also has a specific configuration of artifacts that describe it. The environment contains all the EXTERNALs. The reason metasystems are included is that we are using George Klir's General Systems Theory as our basis for describing systems and it contains explicit meta-systems and meta-structures. Both of those constructs would be described here in terms of metasystems.

A system has an INFRASTRUCURE

composed of APPLICATION, SERVICE, INTEGRATOR, & BACKPLANE. In most instances it is the Application that is of interest. But applications many times work together and need specific software to integrate them; or they rely on other software to provide services; or they need to talk to each other and cooperate even though they are not integrated. So it is necessary to describe explicitly the background relations of applications to other system components. Each of these elements of the system has an IMPLMENTATION and that runs on a PLATFORM which is a particular hardware software combination that allows the implementation to execute. All of these parts of the software system may have their own HIERARCHY.

Multiple design elements may have hierarchies. In fact any design element potentially may have a hierarchy although it is not explicitly represented as such in the language as it stands. It is expected that if the designer needs to express the hierarchical nature of another design element he will just extend the language to do so. The HIERARCHY contains levels which may be arranged in relation to each other in terms of which are ON TOP or ON

BOTTOM of the other. LEVELs may also be adjacent so the hierarchy is not necessarily strict and may be partially ordered. But the levels of the hierarchy have relations to what is know as the GRID. The GRID is a four-dimensional matrix which is meant to hold the generalized design elements. The grid has four kinds of divisions called LAYER, PARTITION, STRATA, and TIER. Both systems and metasystems may have grids. The grid is a mechanism specifically designed to allow humans cope with complexity. It creates a series of places that may be super close packed to simulate the enmeshed quality of most real systems. The parallelism between levels and the divisions of the grid allow us to move from basic sorting of design elements to the actual placement of design elements in mnemonic places with four dimensional addresses so they may assume a specific spatial configuration that mimics their logical configuration.

All the major systemic features that have been mentioned so far may connect explicitly to a particular application domain. Thus we are viewing not just one application but a class of applications, not one system but a class of

systems, etc.

7. DOMAIN

Domain analysis is necessary in order to optimize any class of applications or systems. One looks at the whole class and attempt to create design elements that will be useful over as much of the domain as possible. A domain is made up of GLOSSARY with DEFINITIONS, TAXONOMY, DOCUMENTS, REPRESENTATIONS, PRODUCTS, SPECIFICATIONS containing REQUIREMENTS under CONFIGURATION control. Thus the domain opens up the SYSTEM to a wider analytic perspective on whole classes of systems.

A configuration includes ARTIFACTs which include COMPONENTs that in turn include UNITs. Artifacts may form an AGGREGATION that has a VERSION, DATE, AUTHOR, BUILD or anything else. Aggregations allow the same artifact to appear in multiple contexts during design.

The main part of a DOMAIN is the TAXONOMY which is a classificatory system

of many CATEGORY. Categories contain TAXONs which differentiate elements based on their CHARACTERISTICS. A taxon is a kind of classificational template with the basic set of characteristics that members of the category may have. It is by using the taxonomy that the picture of the systems in a domain is worked up and systems are in turn made reusable and more generic.

8. ARCHITECTURE

ARCHITECTUREs include COMPONENTs that are positioned at INTERSECTIONs in the GRID. Each intersection has it's COORDINATES. Architectures are composed of ARCs and NODEs. Nodes correspond to the UNITs within a component.

Nodes may have INTERFACEs rather than direct connections with each other. This allows for encapsulation of architectural units. A NODE may be any entity within the ISEM sub-languages. An ARC may be any relation between entities. This allows complete flexibility in defining what is in the architectural design. Different applications may necessitate different types of architectures.

There are a series of four entity relationship charts that describe the relations between the different divisions of the GRID (LAYER, PARTITION, STRATA, & TIER) and the INTERSECTIONS. Basically an intersection is an intersection of any three of these divisions.

9. AUTOMATA

An automata is a Finite State Machine. It is composed of a STATEVAR that holds the current STATE. It also may have an associated meta-state called a MODE that actually changes the state machine's behavior in some way.

In the state machine STATEs are connected by TRANSITIONS which are associated with EVENTs and ACTIONs. This allows the state machines to be built up incrementally by one transition at a time. Actions are associated with WORKERS that encapsulate PROCEDURES.

10. ACTOR

The autonomous agencies (separate individual processors of some sort) of a system are represented by three different concepts. An AGENT contains an ACTOR which contains a

TASK. Basically an agent is an abstract autonomous being. An actor corresponds to Agha's definition of 'actor' which is an concrete autonomous executing unit which is not tied to any one processor. A task on the other hand is tied to a particular processor which it is dependent on for resources. Here the OSSWG Real-time System Reference Model is used to describe the levels of operating systems. The lowest level is the LPOS (Local Processor Operating System) which allocated resources to tasks. The IRAX (Intermediate Resource Allocation eXecutive) allocated resources across a field of LPOS units and it would be at this level that actors would be allocated to different processors and moved. In order to be able to find resources, especially when they are moving it is necessary to have a NAMESPACE. The namespace guarantees unique identifiers that may be used as addresses for messages. The POSTALSERVICE is responsible for delivering these messages and making it appear that all message deliveries are local even if communications networks of various sophistications are involved. At the system level we have the SRAX (System Resource Allocation eXecutive) which allocates various

resources to meta-actors called AGENTS. Agents may be a swarm of actors. All actors communicate via messages. But at the lowest level there may be Tasks or Interrupt Sub-routines which have different communications mechanisms. This level is important for real-time system development where performance optimization is constantly a factor. Of course the LPOS is associated with an individual processor, which a typical system different types.

At the task level there is a network of TASKs connected by CONNECTORS linked to specific PORTS. Connectors may be of various types such a QUEUE, FLAG, PIPE, SEMAPHORE, FILE, MAILBOX, RENDEZVOUS for ADA, or just a general multi-party INTERACTION. This is not meant to be an exhaustive list and surely many other communications mechanisms will be added depending on the specifics of the application.

The basic relations between OBJECTs, ATTRIBUTEs and the associated DATASTOREs are also defined in this sub-language. Objects have associated attributes which are kept in datastores. These data

constructs may be read or written.

11. INFORMATION

This sub-language provides a generic way of expressing the content of information. There is a hierarchy where data is given context and becomes information which when absorbed becomes knowledge. At the lowest level we are concerned with the DATUM which may express a myriad of values depending on its type. The generic value representation is the datum which may set the attributes of Entities or Objects based on the values of Expressions. Data values are usually time bound so they are also associated with an interval.

The datum may appear as part of an INFOPACKET which contains many datasets. It is normally the infopacket which surrounds any particular datum that is referenced within a system.

The multiple interrelations between events are portrayed as a series of production rules which based on configurations of events may change other aspects of the system.

12. MACHINE

In this sub-language the correspondences between the different levels of abstraction in the relations between the AGENT and PROCESS viewpoints is made clear. AGENT corresponds to PROCESS. PROCESSOR corresponds to TRANSFORM. TASK corresponds to FUNCTION. The combination of task and function is an EVENTITY. It is the MAPPING of functions onto autonomous individuals which is one of the core minimal methods. The other is the Virtual Layered Machine itself which embodies these mappings.

Both EVENTIES which are seen as the intersection of all four views on the design and PROCESORS (which is to say the LPOS too) has EXECUTIVES associated with them. An executive may either be based on an AUTOMATA or PETRINET. The executive controls a TASK or set of TASKs. The TASK has an associated Virtual Layered MACHINE and a THREAD which contains its allocated resources and keeps track of where it is in case of context switches. The task also has an

associated QUEUE which contains MESSAGEs that indicate which OPERATIONS need to be worked on in what order.

The MACHINE is made up of an INSTRUCTIONSET containing INSTRUCTIONS. Instructions may be OPERATIONS associated with OBJECTs. Both operations and instructions directly implement FUNCTIONS perhaps using the general purpose WORKER to do so. A worker is like a macro which may contain a set of PROCEEDURES. Also a MACHINE may be associated with either an AUTOMATA or a PETRINET which fires the INSTRUCTIONS as their ACTIONs or TRANSITs.

There are a whole series of actions which connect OPERATIONS with OBJECTS and ATTRIBUTES that are described. Also the relations between MESSAGEs that invoke INSTRUCTIONs and come in INFOPACKETS is made explicit.

13. PROCESS

The Process sub-language implements the

Hatley/Pirbhai methodology. In that methodology controlflow is added to the dataflow diagrams developed by Tom DeMarco. CONTROLFLOWs connect PROCESS (the dataflow functionality bubble) and the CONTROLSPEC. Controlspecs may be either PETRINETs or AUTOMATA which drive an ACTIVATOR that turns process bubbles on and off. It needs to be remembered that in the Hatley methodology dataflow bubbles are on and will be activated to produce their outputs unless explicitly turned off by an ACTIVATOR. Before the automata or petrinet may come a decision table to reduce signals to the minimal set.

A DATAFLOW line connects PROCESSes. The envelope of the system is described by a CONTEXT bubble where dataflows connect to EXTERNALs. The dataflow lines may also connect to DATASTOREs which contain OBJECTs.

14. PETRINET

The PETRINET is a network of PLACEs and TRANSITs in which markers move to simulate

controlflow within an application. The petrinet is the dual of the automata (state machine) but it takes many more petrinet transitions to describe a system than a state machine. But in the petrinet the control flow is easily seen ad explicit. There are ENTRY and EXIT places where the markers start out or end up. Since we are describing a colored petrinet it means that markers themselves have structure so that a particular transit must distinguish different patterns of in coming markers. This means that each transit has an associated set of DECISION rules that tell it whether to fire or not. Also the transits are associated with their own transient states which are called BANNERs that contain INSIGNIA. When the DECISION allows the TRANSIT to fire because a recognized pattern of markers has arrived in the input PLACES then the TRANSIT may set the BANNER to an INSIGNIA. These terms have been invented to maintain the distinction between the PETRINETs and the AUTOMATA.

A set of connection operations have been specified for building the PETRINET in an incremental way.

15. SITUATION

The situation describes everything that may take place at a SITE which is a staging area for the construction of ENTITIES.

The ENTITY may have both a BEHAVIOR and an ASSEMBLY. The ASSEMBLY contains its PARTs. The BEHAVIOR represents the response of the ENTITY to higher level inputs. Both the ASSEMBLY and the BEHAVIOR may be specified in terms of PETRINETs or AUTOMATA.

An ENTITY belongs to a CLASS in an inheritance hierarchy. It may have a MODE (meta-state) and has a set of ATTRIBUTES. The entity may be mapped to an OBJECT. Entities are distinguished from objects because the concept of object has a certain narrow range of applicability suggesting encapsulation of persistent data. Entity is meant to be a much more general purpose term. Entities have INSTANCEs which may in fact be PARTs of an ENTITY. An entity may have a FACE which is a set of it’s attributes presented in a particular manner.

The entity may exist at a particular SITE. The site is a non-located INTERSECTION which may later be connected to an intersection in the GRID. Also the ENTITY may have SLOTs that are ATTRIBUTES connected to EQUATIONS that recalculate when any of their inputs change or periodically. The attribute as it appears in an INSTANCE is called a REPLICA.

16. TEMPORALITY

This sub-language implements Allen's Interval Logic in a context where it may be connected to Temporal Logic. Temporal Logic treats modalities such as necessity where one event is necessary because a prior event occurred. A TEMPORALITY is a SHEAF of signals. A sheaf of signals may be further broken down into a BUNDLE of SIGNALs. SIGNALs are modulated lines of input which show variation with time. Within a sheaf bundles may BRANCH off from other bundles of signals. These branches which split and join allow one to reason about necessity within the possible worlds of the system. The ATTRIBUTEs of an ENTITY may be connected to a particular signal and receive its modulation. When this connection occurs we have and EVENTITY.

The EVENTITY is an ENTITY plus a LIFECYCLE whether portrayed as an AUTOMATA or PETRINET. EVENTs may have all the relations between each other described in Allen's Interval Logic such as 'event is before event' etc. Events may be associated with INTERVALs that are defined in terms of two TIMEPOINTs or a TIMEPOINT and a TIMESPAN.

17. TURING

I have named this sub-language in honor of Alan Turing. It contains several key concepts such as ENVIRONMENT, EVENTITY, METASYSTEM, SYSTEM which connect all the other concepts within the sub-languages together. Each of these key concepts have some relation to the SHEAF, BUNDLE, FLAG, EVENTITY, and AUTONETIC. Here we see the confluence of the central concepts of our systems theory. EVENTITYs are active objects that combine ENTITYs and LIFECYCLES. A SYSTEM and a METASYSTEM are composed of EVENTITYs. Or from another point of view they may be seen as special cases of EVENTITYs. The ENVIRONMENT functions in a similar way. In any case an

EVENTITY is an ENTITY that has been temporalized, thus the connection to the SHEAF and BUNDLES of signals or to a CLOCK of it's own. In the case of all these highest level conceptual entities in our systems theory there may be a connection to what is called an AUTONETIC. This word exists as a conflation of *AUTOMATIC* and *CYBERNETIC.* The AUTONETIC is a description of the BEHAVIOR, facing upward in the hierarchy, or ASSEMBLY, facing downward in the hierarchy, like one of Arthur Koestler's 'holons'.

AUTONETIC's contain both BEHAVIOR and ASSEMBLY modules and may even have their own meta-states called FLAGs, these are global variables, that may either be STATEVAR or BANNERS.

This sub-language contains all the necessary calls needed to set the time either directly or by reference clocks. It is assumed that the problems associated with Special Relativity also plague all distributed real-time systems which may not have accurate global reference clocks if the system is too big.

18. WORLD

In this sub-language the relations between AGENTs on different woldlines within a relativistic universe are considered. The general model is Minkowoski's light cones of possible causality. The Microelectronic Computer Consortium (MCC) model called *RADDLE* which was made into a tool called VERDI is taken as the model of the relation of different agents working together in a distributed fashion. The ability for multi-party interactions is taken for granted. A generic description of the operation of an agent on a worldline is called a ROLE in the *RADDLE* language. Roles are made up of constructs that may either contain INTERACTIONs with sets of other agents, ITERATORs, SEQUENCEs, or CHOICEs. The agent cycles through the pattern for the ROLE. *VERDI* the MCC tool allows this to be simulated. Constructs may trigger events which may feed into AUTOMATA or PETRINETs. Each WORLDLINE is generated by a ROLE and may allow the display of multiple SCENARIOs that cut across worldlines that may involve may agents.

19. GENERAL COMMENTS ON ISEM

As you can see, now that we have made a cursory tour of the sets of sub-languages, ISEM is a combination of many existing methodologies. Some of the concepts overlap because as you move from one perspective on design to another the semantics changes. The ideal is to make this overlapping as small as possible. But from our perspective on design it will never be completely gotten rid of so we have interchangeable concepts in the languages with different semantic shading. Also it is somewhat arbitrary in which language some of the statements appear. This is an area where further work needs to be done to determine, given cross links, where any particular statement should appear. The work of narrowing down this language to some minimal and extended set needs to be done, but that must be the work of practitioners as they attempt to apply the language to the design of real-time systems. The language is not meant to be an abstract mathematical image, but rather it is meant to be the tool box of concepts that prove useful to the practitioner. This has an empirical basis. Formalization allows us to narrow down that set to the smallest possible

subset of all the possible design statements.

In ISEM there are certain concepts that do not relate to each other at all. There are others that are highly interrelated. The grouping of the statements into sub-languages should work to get all the highly interconnected concepts in as close a proximity to each other as possible but still allow necessary connections that breach the barriers between sub-languages. When the set of statements becomes well formulated then these interconnections will tell us a lot about the deep structure of the design methods. This is because some elements vanish as you move from perspective to perspective while others do not. If we could ever know exactly how any given design concept is transformed by the movement between perspectives then we would know a lot more about the inner structure of real-time systems.

As has been noted new sub-languages could be added or new statements could be added to any given sub-language. Unlike a programming language extensibility of the language itself is the key. The set of concepts presented here comes from an analysis of existing design methods separating the fundamental concepts

out and adding those necessary to integrate all these methods into a single way of approaching systems design. Other syntheses are possible and encouraged. Designers need to be familiar with all the different methods so they may use them like a tool box in their work. One of the big areas for research is which set of methods is best for a particular application type or design situation.

20. THE FUTURE OF ISEM

ISEM is incomplete, and not fully self-consistent. It needs to be applied to the work of design and rung out in that kind of environment. Once a stable and useful set of statements is produced then there are many uses for such a formalized design language many of which have already been mentioned.

SIMULATION: The language could be used to do dynamic simulations of designs before they are implemented to gain some idea of their performance characteristics.

COMMUNICATION: The language if known by a group of designers may serve as a medium of communication about designs which is sadly

lacking in the Software Engineering discipline today.

REPRESENTATION: We lack any complete representation methodologies today. ISEM strives to be complete. A complete design representation will allow us to record designs and changes in them which should be a boon to understanding systems when their designers are long gone.

EDUCATION: Neophyte Software Engineers may use ISEM as a core around which to build there knowledge of Software Engineering heuristics and experience.

REUSE: ISEM representations may be added directly to the specifications of source code to record the design at a higher level of abstraction. A tool which checks visible identifiers against the ISEM identifiers could ensure that the design and implementations are connected at the most superficial level.

REVERSE ENGINEERING: If you have to understand some one else's design it would certainly help to have a design notation to put what you have learned in so that knowledge

would not be lost again and to jog the memory.

FOUNDATION FOR HEURISICS: What is missing from the ISEM language is the Heuristics that tell the designer how to use the methods to get the best possible design. With a standardize language for expressing design formalisms these heuristics may be stated as rules or policies in terms that are easier to understand.

BASIS FOR GRAPHICAL NOTATION: Once we have some idea what are all the different concepts we need and have some idea about their interrelationships then we can begin to build a set of graphical notations based on those that already are common which encompass all the aspects of design for real-time systems and not some improper subset.

ACCESSING DESIGN DATABASES: Once we have a standard language we can use it to build up design databases that may be accessed using key words and other type of queries.

DEEPER FORMALIZATION: By having a formalization of design elements we can added these to the array of specification formalisms to

get better control of the relationship between the design and the specification.

These are a few of the uses of the ISEM real-time design language. I am sure there are more that may be discovered it we were to use such a system to describe and design actual real-time systems.

21. IMPLEMENTATION OF ISEM

The formal design language presented here is a prototype of a ideal meta-design language. If it were implemented so that it had its own parser it would probably have to change in many ways. An experiment with such an implementation has been attempted by the author. In that implementation it was seen that the BNF of the language has to be structured in a specific way to get the Abstract Syntax Tree to have a certain desirable form. Having parsers is a first step toward meeting the other goals listed in the last section. The value of the prototype ISEM language which has not been fully implemented is that it shows clearly the semantic field which any fromal design language should eventually cover. Finding the best possible way to cover this field must be the

result of further study.

22. CONCLUSIONS

This is a brief introduction to the Integral Software Engineering Methodology (ISEM). An in-depth exposition is really needed in order to show how all the concepts interrelate and to act as a tutorial for those unfamiliar with modern Software Engineering methodologies that have been drawn from to create language that represents a union of many discrete techniques. This union is justified because we need to consider real-time systems from many different perspectives and to move between those perspectives. But all the methodologies are just a tool box for use by the intelligent and creative designer who faces hard problems in the design of real-time systems. There is in real-time software systems an unbridgable gap which can only be crossed in the designers intuition, which is really the gap between linguistic and mathematical types of representations. ISEM is an example of a concrete formal-structural system after the model of those described by George Klir. As such it is a step beyond merely logical or set-theoretical mathematical models into the

domain of software systems which has it's own semantics. We need to specify that semantic field very precisely so we can appreciate that fuzzy, ultimately uncapturable nature of software. It is exactly the fuzzy nature of software, as some one expressed it 'like nailing jelly to a tree', that we need to comprehend so that we may adequately make use of that nature to adapt and fit to the world the systems we build.

KEYWORDS:

Software Engineering, Software Design, Software Methodologies, Ontology, Theory, Being, Philosophy of Science, Technology, Formal Systems, Structural Systems, Systems Meta-methodology.

ABSTRACT OF SERIES:

Software Engineering exemplifies many of the major issues in Philosophy of Science. This is because of its close relation between a non-representable software theory that is the product of design and the testing of the software. Software Engineering methods are being developed to make various representations of the software design theory.

These methods need to be related to each other by a general paradigm. This paper proposes a deep paradigm motivated by recent developments in western ontology which explains the generally recognized difficulties developing software. The paradigm situates the underlying field within which methods operate and explains the nature of that field which necessitates the multiple perspectives on the software design theory. Part One deals with new perspectives on the ontological foundations of software engineering based on recent developments in western philosophy. Part two deals with the meta-methodology of software engineering by laying out the structural relations between various software architectural methods. Part three presents the Integral Software Engineering Methodology formal design language.

OVERVIEW OF PART THREE:

In this third part the Integral Software Engineering Methodology will be explained. It is based on the ontological and systems theoretic foundations laid in the previous parts of this series. The methodology is an example of a complete Software Design methodology

expressed in a formal language which may be used to express software designs at the proper level of abstraction.

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

Draft #1 950710 Editorial Copy.
Not for distribution.

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

[SEFpartISEMa/draft1.1/920824kdp

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