Reworking the Integral System Engineering Method Domain Specific Languages

Kent D. Palmer, Ph.D.

Orange, CA 92856 USA 714-633-9508

kent@palmer.name http://kentpalmer.info

Abstract

My recently completed Ph.D. in Systems Engineering on the subject of Emergent Design[i] focused on the dynamics of the architectural design process. This study led the author to develop certain fundamental ideas that would initiate a refactoring of the Integral Software Engineering Methodology[ii] and transform it into an Integral Systems Engineering Methodology (ISEM), which would then allow architectural design to be applied as an alternative to the UML/SysML paradigm at the Systems level. In this paper we will discuss how the ISEM language has been upgraded and we will propose a rationale for its use in light of the future direction of Architectural Design and how we can define it when we use Domain Specific Languages and Model Based Design. This is part of a larger research program that focuses on using Semiotics to understand Sign Engineering as defined by Pieter Wisse[iii]. We will also highlight the importance of the Gurevich Abstract State Machine Method[iv] and how it can be used as a basis for determining the causal structure and the computability of functional designs prior to the implementation of physical designs that use Minimal Methods for organizing designs into structures that will promote increased performance. In addition, we will also show how the proposed design and method can be adapted to the computing infrastructure on which the applications run. The ISEM language contains most of the UML[v]/SysML[vi] methods plus others, but our focus will be on higher level organizing structures that organize the system level, which will give us a more effective control of the architectural design's structure.

Keywords: systems engineering foundations, emergent design, sign engineering, architectural design. model-based design, domain specific language

Considerations in Architectural Domain Specific Language Design

Domain Specific Languages (DSL)[vii] are gaining popularity but there is a high entry cost for changing the way development is done when using a new approach if we must develop these domain specific languages before we actually do any work to define the Design. The goal of Integral Systems Engineering Language (ISEM) is to facilitate and simplify the application of this type of design approach by providing a template for a textual DSL, which will be supported by tools that every engineer already has. The use of textual rather than graphical tools for design goes against the trend established by UML and SysML. Also, ISEM departs from software languages because it is not closed and context free, but instead is adaptable and changeable to the particular process of the design as it is implemented by the Designer. The ISEM language provides a grammar template by using a spreadsheet that can handle 90% of all necessary grammatical constructions needed for defining static architectural structures. Most General Purpose programming languages use complex syntax in order to represent the dynamics of algorithms, which allows the interleaving of orthogonal language features. But design, especially architectural design, deals mostly with the static structure of the system and normally does not need this level of sophistication. As a result, it lacks the ability to represent any kind of structure that one may wish to produce at the more detailed design level of a programming language. Architectural structures are more narrowly scoped than the features that are now availabe for different types of programming structures that are implemented at the level of software code. We will call this Architectural Design 'staticware' and we would like to represent it at the conceptual level of the ultimate intent of the designer rather than in lower level constructs. Even UML and SysML may be at a level of construct that is too low and too general for the purposes of the Designer. What the Designer wants to do is to capture his intent within the design at a level that more sophisticated users and stakeholders can understand. In addition, the Designer may also want to blend this 'staticware' with design constructs that are referred to here as minimal design methods of the type that appear in UML and SysML. The Designer may also want to use representations with different profiles that are difficult to represent within the domain specific framework of UML and SysML. Textual DSLs can represent anything that can be specified in language and in text. One merely creates an appropriate sublanguage that talks about the given domain in a way that is commonly understood. One of the problems of UML and SysML is their semantic weakness, but the ISEM language is designed to show multiple connections between elements at the same time rather than merely using only two-point connections with lines. ISEM also supports 'knowledge capture' at the conceptual level at which the Domain Specialist is thinking and this is subsequently augmented by the design concepts that are taken down to the detail where software code can be written or generated. Knowledge capture not only supports knowledge management but also supports Ontological Engineering because, in the process of creating the Domain Specific language, the ontology of the domain is recorded as well as the context and other dimensions of the problem, as well as solutions that cannot be captured easily by general purpose design methods and languages.

ISEM language appears as a set template of columns in a spreadsheet. It comes populated with general purpose DSLs that show the structure of the language statements in many examples. But, the advantage of ISEM is that the Designer can not only make up his own statements that will extend existing languages, but he can also create his own domain specific language when using this template. This process has two steps because after the requirements development is completed, then the Architectural Designer must begin to think about the specific domain concepts that need to be represented in the design as well as the architectural concepts that could be used to structure those concepts in the particular application that would implement the requirements. The key to ISEM is the flexibility of the language template and the ubiquity of the supporting tool that lowers the barrier for entry into this approach. The process of Architectural Design could produce domain specific languages and extend existing design languages that express minimal design methods. The Architect would use the statements at the Model Level to produce Instance Level Designs for different parts of the system. These languages that form ISEM can be mixed together as necessary and be added to new languages that express domain concepts. They are expressed in text that is initially produced according to the grammar template in the spreadsheet. However, the text could be exported into files and managed in configuration management systems and manipulated by text editors. Spreadsheets would be the best way to easily manage and manipulate the statements of the language because their use is quite natural and also allows an analysis of the languages, the models, and the scripting that can manipulate the models.

We are very interested in the fact that it is much easier for someone to make up their own language than it is to learn the language of someone else's. This fact is exploited in ISEM where there is freedom to make up Domain Specific Languages at will, and to change the syntax by adding columns to the spreadsheet template. This means that the Designer is not hobbled by a preexisting language that was created by someone else. The languages of ISEM are examples that can be leveraged to see how the languages work, as well as how these languages should be constructed on the form of the ISEM template. Also, they can be used as a 'tool box' to suggest the kinds of statements that might be needed for the design of realtime systems, while the architects are free to make up their own languages and statements and to create identifiers for instantiating existing statements. This means that the Architect can remain fluent and be in complete control because of the overall understanding of the languages that he is using since he has created them (for the most part) to satisfy the requirements of the domain that he is addressing.

The Structure of ISEM

ISEM has been created to be as easy to read as possible. Each column in the spreadsheet has a place for a particular grammatical lexical unit. Reflective language is the first language in ISEM and it describes ISEM (itself) as well as all other ISEM languages. The second language in ISEM is a representation of KM3[viii] from the INIRA[ix] ATLAS group that has simplified the description of the models in all meta-levels beyond what is available from the OMG in UML infrastructure as well as Meta-Object Facility (MOF), or Eclipse Ecore, or Object Constraint Language (OCL). Using the KM3 language representation and the ISEM reflective language allows all possible languages to be described regardless of their complexity and domain. However, since we wish to advocate the use of the Gurevich Abstract State Machine (GASM)[x] Method and the Wisse Metapattern Methods[xi], there are languages associated with these methods as well as a State Machine Method that realize the same structures that are available for state machines in UML. The Gurevich Method is a set of rules that allows computational systems to be created from language constructs in any domain. The Wisse Method aids the creation of contextual objects rather than normal object-oriented constructs of entities that are free of context. Taken together, the Wisse and Gurevich methods complement each other as a way to generate the structure of the system at a high level of abstraction. This makes it possible for one to adapt this model to any level necessary in order to articulate the functional and causal structure of the system so that it can be designed in a way that shows that it is computationally viable. However, these models are not concerned with performance. It is only after the introduction of performance issues that it becomes necessary to introduce design concepts for separating and organizing the natural functions and hierarchies of the system. At that time we would introduce other minimal design methods to augment the domain specific languages and the state machines and rules that have been developed in the bridge models, and this will allow us to connect between the requirements and the performance architecture of the design. If there were not performance issues or cross-cutting concerns, or a need for object inheritance, then we could merely continue to elaborate the GASM model until we had a running simulation of the system under design. But, for practical purposes we only elaborate the GASM model until it is clear that the functional and causal structure is coherent, and then, at some point, we could transition to performance oriented Architectural Design, which is at a higher level than Detailed Design. However, it is important to note that in this transition the very models that we have used to articulate the system (up to this point) can continue to be elaborated with performance oriented changes. Furthermore, if it is refined properly, architectural design can serve as the basis for various levels of code generation as long as it is properly augmented by minimal design method extensions. The real promise of a DSL approach is that it allows the various model meta-levels to be used as a basis for code generation, while at the same time capturing knowledge at each level of the design.

The original ISEM had a real-time software orientation and had a complete set of minimal methods that were necessary for producing real-time systems. This was created prior to the introduction of UML. The new ISEM is slated to borrow from UML wherever there is an established standard. But it still represents the same set of minimal methods that are necessary for real-time design and departs from UML wherever it is necessary to extend the language to make it work more efficiently for real-time designs. UML existing profiles have the necessary features for the design of many different kinds of systems and there is no reason to introduce unnecessary disparate terminology now that a standard exists. But, the UML profiles that exist are limited in their expressiveness and they do not have all the profiles we would like to have for supporting real-time design completely. However, the thrust of the new language is not in the area of minimal design methods, but rather at an architectural level where new concepts of a higher order than UML and SysML are being introduced. That is where the genuine novelty of the new ISEM language lies because it introduces a combination between the Ossher GRID[xii] and the Lano N2[xiii] methods, which allows it to fully flesh out the architectural design level of the ISEM languages. The first version of ISEM used Ossher's GRID in a four-dimensional form to produce a model of architecture that could bring together all the various minimal design methods to describe an architectural component. But the new ISEM language elaborates on this level of abstraction that contributes to developing a way of expressing Emergent Architectural Design.

The Basis of ISEM

The author recently completed a Ph.D. research project that developed the concept of Schemas Theory while studying the foundations of Systems Engineering. Schemas Theory provides a basis for understanding the nature of the Emergent System and Meta-system (Operational Environment) Design, and it is on the basis of this research that we now return to the definition of the ISEM language so that we can express (in a fundamental way) the representations necessary to capture an emergent architectural design. ISEM can capture these representations because it allows the Designer the greatest possible freedom to create new statements in any language and to create new languages that can express the design intent of the System Architect at the highest conceptual levels. We base our method on Wittgenstein's Tractatus[xiv], which defines the basis for how we may allow any fact about the design to be captured either with an existing statement or a newly coined statement. The Designer can even change the grammatical template of the languages as he sees fit by adding columns to the spreadsheet and labeling them, which then enables him to use that new grammatical feature. So, neither methods nor tools will hinder the creative designer from creating the most accurate and domain specific model of the system. Because the language template is simple and because it is a combination of operator/operand and Subject/Verb/Object it can style the expression of intent within the framework of the language template in a straightforward and flexible way. 'Define' statements create the ontology of the application and 'Posit' statements relate those ontological statements to each other not only in inheritance or ownership hierarchies, but in all other ways that might be needed to express what is necessary within the design. Lemmas and Operations can be added as necessary to produce dynamic representations of the system. All four meta-levels of the model structure can be easily expressed using KM3 languages. However, the spreadsheet template grammar expresses the meta-model[xv] while the statements' generic tokens for identifiers express the generic model level. When identifiers are added they express the terminal model or the instance model level. KM3 would be used to describe the *meta-meta-model* of the system. KM3, which is a self-reflexive meta-meta-model, is unlikely to change for any reason because it expresses the most fundamental ontological constructs such as the definition of entities and relationships. The grammar of the ISEM template could be changed occasionally, but this should be done only after a great deal of consideration. The actual models expressed in that grammar could be completely reworked, if not reinvented, as well as new languages added during the design process. The Designer does not have to stick with the statements as presented in the language. Any statement could be changed easily to suit the designer's style of expression. Of course, these changes should be coordinated and filtered when there is a team involved in the design. Design teams should use statements of the same style to avoid confusion, although this is like any style guide. As soon as a language is changed, then the consistency and completeness, as well as the well-formedness (clarity) of the language, becomes subject to analysis, which will confirm the correctness of the language. Unlike languages that are parsed, it may be that the Designer will allow the language to remain para-complete, para-consistent (See Graham Priest[xvi]), or para-clear. In other words, keeping the language consistent, complete, and clear may be a second priority to the actual representation of the design in the heat of the moment. However, if the language itself is complete, consistent, and clear, then that assures the quality of the terminal or instance models that are created from the statements in the generic model. If the generic models are allowed to become incomplete, inconsistent, or not-well-formed, then some sort of model transformation will be necessary. However, if the Designer is consistent in his use of the new statements that he invents, then much of that work can be done by 'search and replace' within the spreadsheet. It is fairly easy to rework the model so that it is complete and consistent (again) and that well-formedness is taken care of by the spreadsheet format. The advantage of ISEM lies in the fact that the Designer can make up new statements on the fly as he needs them without generating a parser or doing model checking and this will enable him to capture the design concept in a more effective and expeditious manner. After the design capture has been done, then the Designer can devote himself to making the final product more complete and consistent after the fact when it matters most. In this process the Designer becomes his own Methodologist and is also freed to express the central concepts of any domain as he understands them and in a manner that others can also understand. This is because there are no unnecessary syntactical encumbrances such as those that appear in pre-created languages. The spreadsheet makes it easy to create all sorts of concordances of the language and to search for language sentences to express what needs to be expressed, although, if you don't find them quickly you have the option of creating something on the spot and need not worry about placing it in a normative form until later. Since the language mimics simple English (or the operator/operand structure) it is easy for the Designer to make up simple and understandable statements that express exactly what he wants to express. He can worry about reconciling these various statements' styles that capture the same intent later. However, even though the statement structure is not very important, using the same identifiers for the same thing is very important. Search facilities and concordance facilities (provided by the spreadsheet) can give some guidance as well.

For Systems Engineering, this new approach using ISEM should simplify the life of the System Engineer. Now he can write textual explanations of processes that are difficult to manage. Just as we learned to use single axiomatic sentences for requirements, we must now learn to use DSLs to express the Emergent Architectural Designs that we create in order to reflect those requirements in the implementable system design. Of course, we should still have textual descriptions like the Operations Concept (OpsCon)[xvii] and specifications that are explanatory, but the base design representation needs to be expressed in textual DSLs texts instead of natural written language documents and these would be easy to control with textually oriented configuration control systems. And, they will help us to do estimates in the future because it will give Systems Engineering the equivalent of a line of code (SLOC)[xviii] for the purpose of measuring the productivity of our systems design. Because the tool that is being used is actually a spreadsheet, it is already readily available to every systems engineer who has a basic computer desktop set up with Open Office or MS Office. We can now track how long it will take to create a generic DSL, or to produce a terminal, or an instance model by using a DSL generic model.

The Meta Core Sub-language

The first 'META Core language' would be a reflective language derived from a template for language syntax. This reflective language will make it possible to have a meta-language about any language that one creates within this template structure. It deals with the parsing of the language and its structure, which is composed of meta-information, instructions, and a grammatical SVO (Subject Verb Object) template. The structure can be parsed into a tree of terminal and non-terminal symbols that can be turned into an abstract syntax tree of operators and operands. The language allows the speciation of the sequence of terms in the language as it appears in the headers of the spreadsheet. This meta-language called "META Core language" is distinguished from the "META KM3 language" that allows the description of all models at whatever meta-level. KM3 is a simpler version of what appears in the UML infrastructure and the Meta Object Facility (MOF)[xix]. There are hierarchies of both meta-languages and meta-models and they are different from each other. One is *the representation* and *the other is the conceptual structure that is being represented*. Both have a series of meta-levels that are parallel to each other. The META Core language is just one representation. There may be many representations of design concepts, for example, graphical representations in UML. It is interesting that *programming* is *textual* but *design tools* have become *graphical*. The problem with this is that the graphical languages such as UML and SysML are semantically weak. Even though ISEM is very simple, it is still much stronger semantically because it can have statements that relate multiple objects with each other at the same time. ISEM is also represented in simplified English and it is readily understandable. There is nothing to stop someone from creating a graphical representation that mimics the domain specific languages that are created. The problem is that this step is complicated and difficult to do as a prerequisite to creating the design. The problem with UML and SysML is that they already exist as tools, yet, they are generic (like generic programming languages) and thus are difficult to extend for the purpose of producing domain specific profiles. Here we are trying to achieve a first step that will allow anyone to create DSLs with the tools they already have and thus derive the benefit quickly so that they can see if it is worthwhile to produce parsers for their own DSLs, or new Profiles for UML, or other types of more complex approaches to using DSLs in development. In other words, using ISEM does not preclude the later use of more complex implementations of Model Driven Design and Domain Driven Design[xx] using Domain Specific Languages, but it offers a first step where the *value of the DSL* can be proven to the Systems Engineers and Software Engineers concerned with Design. We especially want a tool that is easy to use so that we can facilitate the Emergent Designs of new and innovative systems that will not get in the way of the design process itself or require retooling in order to reap the benefits of Model Based Design and Domain Specific Language development and use.

Diagrams and Models

Much design today is done through using Powerpoint slides or Visio diagrams. These diagrams are semantically weak because they only indicate what is being said by the person who developed them for presentation, and they only retain their meaning as long as that person is around to explain the diagrams. If, on the other hand, we take these "Powerpoint engineering" diagrams and produce a *domain specific language* that explains what their objects and lines mean *then we will have captured a shared and transferable knowledge in a reusable medium*. It is quite interesting that from the time of Euclid[xxi] our tradition has relied on diagrams and text for the construction of proofs and that even today this is fundamental to engineering. Look at any specification and you will see text, diagrams, and tables for the most part. But there is no *meta-information* that structures the *meaning* of the elements in the text, diagrams, and tables. *Thus, these descriptions have no models behind them but only have a surface description, which we rely upon to describe and explain the system*. Without the models there are no proofs, i.e., no closure to the descriptions and explanations. Having models allows us to prove properties about the design such as their consistency, completeness, and clarity, which are the properties of all formal systems. If a model is para-consistent, para-complete, or para-clear then we can do analysis to find out where the gaps in the model lie. This is not possible when there is merely a *surface* representation by text, diagrams, and tables with no models behind them.

In the geometry of Euclid there is a parallel between the diagram and the statements of the proof. This parallelism allows the analyst to see that the relationship between the two is isomorphic and closed, which allows the proof to be conclusive. Models offer this possibility for engineering systems. *If we had models that are traced to our requirements behind our specifications, then we could use those models to connect things that appear in diagrams, texts, and tables within our specifications.* Models are simple to create if we use an approach like ISEM, although it is difficult to confirm their proprieties by analysis. However, in normal engineering systems we do not prove the properties of designs but merely trust the due diligence of the Designer. It is important to understand that the model only has the properties of a formal system, i.e., consistency, completeness, and clarity. These are the properties that appear between the Aspects of Presence, Identity, and Truth. *It is only when we add the Aspect of Reality that we get the crucial aspects of verifiability, validity, and coherence*. Verifiability is the relationship that the model has to the requirements. Validity represents the relationship between the model and the context of use. Coherence is the internal synergy and integrity of the model that is achieved when we add a model behind the surface presentation of a system design representation. For example, it is possible to pose questions about the design model that underlies the representation when we use an MS Powerpoint slide diagram that has been annotated with the modeling statements in a DSL. The Powerpoint diagram cannot answer our questions, only the Designer can satisfy our queries, particularly if the slide is not annotated. As Peter Naur[xxii] points out, no amount of documentation can capture what the Designer has in mind, but a model can serve as a representation of the knowledge that his diagrams and pictures do not capture. We can ask if the model is complete and consistent with itself, and if it conforms to the well-formedness rules of the representational medium. We can also compare the terminal or instance statements of the DSL and view them as 'taken together' or as a synthesis of the requirements in the actual context of use. This is easy if the language can be understood as a stakeholder that generates the requirements as 'needs' or 'wants', and if the end users in the operational environment can also understand the requirements and how their operationalization of the system can be used in its final place of deployment where it will be used as a tool to help them accomplish their goal. By triangulating the requirements through verification and the operational environment through validation, we can discover *the coherence* of the model, which is a deeper property the model has within itself because it signifies its synergy, integrity, and poise toward the environment beyond its own synthesis. Without the model it is impossible to query surface representations concerning these properties because the surface representations cannot be shown as closed. Without the model we are assuming that the conceptual model is in the designer's mind. The model allows us to visualize the Designer's conceptual representation of the model. This allows others to look at the model and to analyze it. Furthermore, the model can become a vehicle for the individual to capture and communicate his tacit knowledge, which is informing the design indirectly. By creating a model we can represent knowledge directly and then use that knowledge to test the coherence, validity, and verifiability of the surface representations of the design.

Picture, Plan, and Model

In my dissertation I discuss the sub-schemas of Picture, Plan, and Model, which are projected as precursors of the Whole Form that is being designed. The diagrams that appear in Powerpoint designs are pictures. They are normally global pictures of the parts of the design, or they may be pictures of the different parts of the design. In order to produce a model one must first have a plan. The plan (like a blueprint) shows the object under design from different viewpoints with the various features measured and augmented by notations. The plan allows us to know what would be repeated in different dimensions to produce the model. The model is usually an abstract representation of the whole system in miniature, sometimes with certain details omitted, or in some cases, as a yet non-working configuration that abstracts from the complete functioning of the final synthetic system implementation. We are advocating that we should use the ISEM language to produce plans and models that would stand behind our diagrammatic representations with the same integrity as the statements expressed in a Euclidian Proof (that are isomorphic to the diagram of the proof). To produce the plan we must construct a series of models, meta-models, and meta^2-models that form the basis for the representation of the *terminal* or *instance* model. A series such as this can define what will be repeated in the representation and can be used to build up the terminal or instance model. It defines the ontology of the domain by specifying the possible categories and elements that can appear in the model. Then, a series of meta-models will describe what kinds of relationships that these ontological elements can have with each other. These are stated in a restricted English formalism. They are posited relationships between ontological elements identified through definition statements. These posited relationship statements can relate the elements to each other in order to make sense within the domain. Beyond these two basic statements we can also have Lemmas and Proof type statements. Lemmas are steps toward proofs; they include the relationships between statements that establish deeper structures in the modeling system. Of course, we would like to prove our designs and that is (for the most part) a distant goal. But, at this point, what we *can* do is to build up Lemmas in order to strive for a kind of closure that would allow proofs to be written about the system. We should think of the system as a synthesis, and if that synthesis is completely closed, i.e., a convex solution, then that is what we will ultimately strive for in our models of the system. We are in a design process that posits synthesis and works toward it. We move back and forth between the analysis of the model and its representations, as well as the further elaboration of the synthesis that we are projecting on the design. Charles Peirce[xxiii] calls this property of the synthesis a "Third" because it has the assumed continuity of the design as a whole system. The individual objects and their relationships are the "Firsts" (Isolata) and the "Seconds" (Relata) in the terminology of Peirce[xxiv]. Peirce makes a very interesting point that is not normally understood, which is, that there is a difference between the *precision* of analysis and what he calls the *precission* of the synthesis[xxv]. Precission, with two '*ss*' means that we are not taking the system synthesis apart, but we are articulating the various aspects of the synthesis when it is still operative. When we produce models we need to oscillate between the precision of Analysis and the *precission* of Synthesis as we approach the limit of the 'system synthesis closure' that we want to achieve in the design. But, if we do not have a model representation of the design, this task is very difficult to perform. *What we want is a representation that will give us the most direct access as possible, and that is what ISEM attempts to give: a representation where the structure of the syntax is as simple as possible and a representation where the formal language is as direct as possible so that we can oscillate between the precision of Analysis that deconstructs the parts of the system and the precission of Synthesis that works backwards from the posited synthesis.* A representation such as this will give clarity and soundness to the design and will facilitate the shared vision of the design team. We want reviewable documents that will represent the model of the design and capture the knowledge of the system at the *level of the intention* of the users and stakeholders of the design. In other words, domain experts should be able to read the representation and understand it without being hampered by the clutter of the syntax of computer languages. It should represent the concepts they are working with and not some transformation of those concepts into another language or representation that is familiar only to the system architect and the software engineers. ISEM allows this oscillation between the decomposition of analysis through precision and the articulation of the projected design Synthesis by precission. The design synthesis we are projecting, which is emergent, is our vision of how to build a system so that it actually works as intended for the customer and end user. This demand causes us to create new statements in our generic design model languages and domain specific languages. It may also drive us to create new languages to capture other views of the system that are either domain specific or necessary due to the technological infrastructure we are using to realize the system. The languages that represent the models themselves are syntheses at a higher level of abstraction in the knowledge domains that support the specific system design. Many times these knowledge domains remain tacit within the kinds of specifications we normally use to describe the systems that we build. ISEM type modeling languages attempt to make these modeling domains explicit and thus can be a means for capturing our own knowledge about various domains, or for communicating that knowledge to others in a concise form. It is a way for us to collect our own knowledge of design and to develop our own architectural styles. Designers could produce their own languages with a language design that supports their own knowledge, styles, and approaches toward their design. ISEM is merely one example that shows the feasibility and efficacy of such a simple and direct approach to Model Based Engineering and Domain Engineering that includes an example version of Domain Specific Languages tailored for architectural design. Every language could be omitted, tailored, and recreated as the Designer (acting as domain engineer) saw fit. The high entry costs of designing parsers or graphical tools for our domain specific languages are not necessary. We can benefit from Domain Specific Languages in these Model Based Designs quickly and efficiently by using the tools we already have in a new way. This allows us to experiment and prove our concepts before we invest in expensive tooling that we may not use, as well as avoiding the wasted time and energy spent on an approach that has been developed in a traditional manner that uses only the surface representations of our designs. The promise of model based design will give us the opportunity to use these models to generate products from our design, such as code. *Eventually we want our models to become the center of the system rather than an appendage to the system.* As long as the code or the material implementation of the system is the center focus, then we are stuck producing documentation that may never be used or could possibly be wrong. However, if we eventually generate the code or material implementation from the *model*, then we will continue to update the models because they will become the center of the system implementation rather than an unnecessary supplement.

The Importance of Parsers

When we finally arrive at a point in time when there will be a parsed language for the ISEM DSLs, we expect that the Domain Specific Languages will have become somewhat standardized so that new parsers only need to be introduced for handling new statements that are added by the Designer rather than building new parsers for every variation of the languages. Presently, parsing is deferred in these DSLs because our adherence to 'parser restrictions' limits the creative expression of the Designer. Rather, the emphasis is on maximal expressibility at the point where the Emergent Architectural Design is being visualized and synthesized. At that point we want to have a platform of a language template that we can use, although we want maximum resilience, adaptability, and flexibility in our mode of expression so that we can simultaneously be developing the representation for the design and the design itself. Our language does not have to be able to express everything from the very beginning as programming languages do. Rather than having a closed and context free language, we want an open and 'context sensitive' language for our first attempts at capturing the emergent synthesis of the design. Later, once we have a sketch of the design at some level of abstraction, then we will be able to fill in that synthesis through an analysis of the language extensions or new domain languages that we have created in the process. At a later stage we will want to formalize the languages that we are using and tighten them up by improving their formal properties. At that time we can refactor our designs based on a higher formalism and introduce models whose representations can be parsed and whose conceptual systems have the necessary formal properties we desire. Thus, we are advocating that we allow for para-consistency, para-completeness, and para-clarity in our languages and models during the first part of the design process as we create an infrastructure that supports the design and develops along with the superstructure of the design itself. This will allow the architectural design representations and models to be fully emergent.

As a model based engineering approach, ISEM embodies the technologically simplest possible way that an entire hierarchical structure of meta-models and language representations can be easily understood and easily manipulated. There are many complex standards in this area, but these complex standards get in the way of understanding the fundamental ideas behind Model Based Engineering and Domain Specific Languages. With ISEM, it is possible to see a modest yet fully integrated approach to Model Based Engineering with DSLs. It uses tools that everyone who operates within desktop environments will understand. It does not require any special tools that cost money beyond the bare minimum that has already been spent to support engineering work. It allows products to be produced that can be reviewed, shared, and configuration managed. The models can be incorporated into standard engineering products such as specifications and operational concept documents. It does not use any unreadable syntax such as XML as its basis, but can easily be transformed into XML if necessary to interface with other established or future representational tools. Model transformations could be used to move the designs that are in ISEM into other tools. *Thus, designers can produce products on a small scale and then transform them into a larger representational environment as necessary.* There are also other textual representations of models being developed, for example, Xtext[xxvi] is part of Eclipse and could be used interchangeably with ISEM or as a transformation of ISEM. The parsing and transformation of ISEM into other graphical or textual representations is always possible. Having a parser for ISEM is not crucial because it can easily be built with ANTLR[xxvii] and other parser generation systems if necessary. It can also be embedded in extendable languages such as M, Converge[xxviii], or PI[xxix]. When we first attempt to capture the Emergent Architectural Design it is more important to maximize flexibility, adaptability, and resilience while the vision is still fresh. First, we may write down a description of the system design as a brief sketch. We could make drawings on the back of envelope (or on a quadpad). But then, as we articulate and solidify that design, we may want to describe it more formally in simple English by using statements that capture what we know at the time. ISEM is designed so that the various facts that you know about the design can be captured in a statement. For this reason ISEM has more statements that are absolutely necessary because it has statements that will allow each entity and each relationship to be captured independently. For instance, if we only know that one state exists we should be able to write that down, or if we only know that there is a relationship between two states we should be able to capture that also. But eventually we may want to move toward a more concise representation where there is a single statement for each state vector in the system. ISEM allows you to discover the nature of the design as it emerges and then to consolidate the various features of the design as more information is gathered that allows the designers to make that consolidation. It allows you to create new statements or whole languages for capturing the novel and emergent qualities of the design. It is this simple extensibility of ISEM that makes it valuable as the first responder in the design process. It takes the designer’s knowledge and makes it more concrete and applicable to the design as it emerges. And we know that the process of writing allows us to discover new ideas and expand upon our design paradigms as we design. This then allows us to apply analysis and synthesis to the representations that are created in ISEM as well as to consolidate what we have learned in the emergent process of creating the first architectural design model. Most designers study methods, languages, tools, and other applications in order to form ideas about how things can be designed. Instead, we engineers tend to make up things as we go along when we are engaged in the design process and we use methods and tools in an ad hoc fashion, first, by doing everything by hand (possibly on quadpads or on pieces of paper). We then take these thoughts and ideas and refine their sketches until we have something workable and *this is the point when ISEM proves to be very useful*. ISEM allows these sketches to be captured in the first formalism so that consistency and completeness checks can be done early on the sketches.

Terminal or instance models may only use a small part of the available language but they are only as complex as the system that is being designed. As the design progresses we could reduce the number of statements that are being used to express a feature of the design. For example, we could replace separate statements about states and transitions and functions with singles statements about state vectors. So, there is some expectation that there would be consolidation of the instance or terminal model as the design progresses. Also, there would be an analysis of the completeness and consistency of the design concept that the terminal model is expressing and this analysis could be folded back into the ISEM representation. As a result, the design description would improve over time and would provide a good departure point for the next iteration of the design process. When we begin to sense that our design is sound enough to share, we can then create diagrams and charts to share with others using the standard tools. *But, behind those diagrams and charts you will have the support of the ISEM representations that formalized and captured the knowledge that is contained in them.* Eventually we may want to use a tool that is standard for large scale design representations within our engineering environment. If that is the case we can then transfer what has been semi-formally captured in ISEM into that particular tool either by writing a parser and transformation tool in order to use the text, or by manually re-entering the information into an alternative representational format. Those tools normally have a frozen formalism that must be adapted to in order to use them and they may not capture all the aspects of the design that came out during the design process. So, a case can be made that even if you are going to use UML or SysML to capture the generic parts of the design, you may still want to use ISEM to capture technical details or domain specific features that those tools do not support beyond the traditional methods that we have become accustomed to. Even if you were to develop your own UML profile you may still wish to try out ideas that may express how that profile should be structured before you spend the money to build the new profile for the UML or SysML tool you are using.

Changes to the Languages

As a language for software systems architectural design, I think ISEM has value. So the point of this research is to bring the language up to date with the latest research on Model Based Design and Domain Specific Languages. Now that standards exist in this area there is no reason for not using those standards as the basis of the language as long as the standards are not so complex that they obscure the fundamental ideas being expressed. For example, I was lucky to find KM3 as a basis for meta-model description because the transformation of the UML infrastructure and the MOF was a daunting task. Fortunately, that complexity was not needed. I now begin with the *reflexivity* of the language as my starting point so that all the languages that are written in ISEM are reflective[xxx]. One really only needs a reflexive language definition and the KM3 representation in order to be able to start creating one's own languages as an alternative. This was a useful learning task and a worthwhile 'knowledge capture exercise'. As we try to create model languages we begin to understand how much we know about the minimal methods that they employ. It is one thing to have a general idea of how these methods work and another to have a complete consistent representation of the minimal methods. The minimal method languages will produce a model that can be used as a basis for building other languages in other domains using standard domain engineering techniques.

Ultimately, applying ISEM to a problem will provide us with a methodological representation of the projected design that will more inclusively represent the 'problem domain' where other solutions can also be represented. This allows us to have a basis for exploring the design landscape within the design process. ISEM supports the possibility of sketching many different designs within a problem space and then allows us to compare them in order to find the best option. Exploration of design landscapes is difficult because representations are difficult to change. ISEM allows you to pick your level of abstraction and then work variations (as necessary) to explore the design language by merely copying a given design fragment and then changing it. By producing several different design fragments at some level of abstraction, then one can do concrete trade-offs between them because the essential differences are represented explicitly in the various design fragments. ISEM models can also be used as a basis for simulation. It is a representation that can be read as a configuration file by a hand-built simulator to actually test the dynamics of the proposed architectural decision. If the modeling system takes on another form of representation in order to drive the simulation, then model transformation techniques could be used to make that transformation between different representations demanded by the simulators. But, of course, if one is building a simulation oneself then it can be tailored to ISEM and the languages that one has built within ISEM. ISEM has been created so it can be easily parsed. Identifiers always come directly after their entity type. Also, ISEM is a simple language and it is only necessary to parse the small part of ISEM that is actually used. Parsing techniques are well understood, but model based design and DSLs are new concepts that can provide a simple means for systems and software engineers to use in order to see what they can do to improve the efficacy of the design process itself. The enforcement of the language can be added quite easily by anyone who desires to have their version of ISEM enforced. Yet, enforcement of any given variation of ISEM could be a difficult problem when faced with extending the language or creating new languages without parser support. We assume that languages such as M from Microsoft Research[xxxi] that are part of OSLO[xxxii] as well as MS SQL server and other academic languages like Converge and PI will eventually support an easier incorporation of textual DSLs. So, we will allow that research to mature while we work on improving the base language. We would like to see an open source version of M, which is itself an open language that incorporates most of the features we need and ISEM can be augmented with OCL[xxxiii] or Eclipse Ecore[xxxiv], which would also ease the constraints on its use.

References on Model Based Design and Domain Specific Languages

Backhouse, Kevin. Abstract Interpretation of Domain-Specific Embedded Languages. , 2002.

Balasubramanian, Krishnakumar. Model-driven Engineering of Component-Based Distributed, Real-Time and Embedded Systems. , 2007.

Börger, Egon, and Robert F. Stärk. Abstract State Machines: A Method for High-Level System Design and Analysis. New York: Springer, 200

Boström, Pontus. Formal Design and Verification of Systems Using Domain-Specific Languages. Turku: Turku Centre for Computer Science, 2008

Czarnecki, Krzysztof, and Ulrich Eisenecker. Generative Programming: Methods, Tools, and Applications. Boston: Addison Wesley, 2000.

Domain-specific Languages: Proceedings. Berlin: Springer, 2009.

Dykman, Nathan. Utmf: An Agile Approach to Domain Specific Modeling Languages. , 2010

Eco, Umberto, and Thomas A. Sebeok. The Sign of Three: Dupin, Holmes, Peirce. Bloomington: Indiana University Press, 1983.

Evans, Eric. Domain-driven Design: Tackling Complexity in the Heart of Software. Boston: Addison-Wesley, 2004.

Fowler, Martin. Domain-specific Languages. Upper Saddle River, NJ: Addison-Wesley, 2011.

Friedenthal, Sanford, Alan Moore, and Rick Steiner. A Practical Guide to Sysml: Systems Model Language. Burlington, Mass: Elsevier/Morgan Kaufmann, 2008

Ghosh, Debasish. Dsls in Action. Greenwich, Conn: Manning, 2010

Giese, Holger. Model-based Engineering of Embedded Real-Time Systems: International Dagstuhl Workshop, Dagstuhl Castle, Germany, November 4-9, 2007 : Revised Selected Papers. Berlin: Springer, 2010

Grant, Emanuel S. Defining Domain-Specific Object-Oriented Modeling Languages As Uml Profiles. , 2002

Gronback, Richard C. Eclipse Modeling Project: A Domain-Specific Language (dsl) Toolkit. Indianapolis, Ind: Addison Wesley Professional, 2009.

Gurevich, Y. "The Abstract State Machine Paradigm: What Is in and What Is Out." Lecture Notes in Computer Science. (2002): 24.

Kelly, Steven, and Juha-Pekka Tolvanen. Domain-specific Modeling: Enabling Full Code Generation. Hoboken, N.J: Wiley-Interscience, 2008.

Kerzhner, Aleksandr A. Using Domain Specific Languages to Capture Design Knowledge for Model-Based Systems Engineering. Atlanta, Ga: Georgia Institute of Technology, 2009.

Kleppe, Anneke G. Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Upper Saddle River, NJ: Addison-Wesley, 2009.

Knopf, Markus, and Sally S. Slish. Domain Specific Languages. , 2001.

Lauder, M, S Rose, A Schurr, and M Schlereth. "Model-driven Systems Engineering: State-of-the-Art and Research Challenges." Bulletin of the Polish Academy of Sciences: Technical Sciences. 58.3 (2010): 409-421

Mernik, Marjan, Jan Heering, and Anthony M. Sloane. "When and How to Develop Domain-Specific Languages." Report / Software Engineering. 309 (2003).

Metry, Alex. Validating Internal Domain Specific Languages. , 2009.

Model Driven Engineering for Distributed Real-Time Systems. Hoboken, NJ: John Wiley & Sons, 201

Nicolescu, G, and Pieter J. Mosterman. Model-based Design for Embedded Systems. Boca Raton, FL: CRC Press, 2010.

Parr, Terence. Language Implementation Patterns: Techniques for Implementing Domain-Specific Languages. Lewisville, Tex: Pragmatic Bookshelf, 20

Parr, Terence. The Definitive Antlr Reference: Building Domain-Specific Languages. Raleigh, N. Car: Pragmatic, 2007.

Petriu, Dorina C, Nicolas Rouquette, and ystein Haugen. : Model Driven Engineering Languages and Systems; Model Driven Engineering Languages and Systems; Models 2010. Berlin; Springer; c2010, 2010

Priest, Graham, Richard Sylvan, Jean Norman, and A I. Arruda. Paraconsistent Logic: Essays on the Inconsistent. München: Philosophia, 1989

Priest, Graham. An Introduction to Non-Classical Logic: From If to Is. Cambridge: Cambridge University Press, 2008

Roberts, Maxwell J. Integrating the Mind: Domain General Versus Domain Specific Prosesses in Higher Cognition. Hove: Psychology press, 2007.

Rössler, Wolfgang. Model Driven Engineering for Safety Relevant Embedded Systems: Model Based Code

Generation for Automation Systems. Saarbrücken: VDM Verlag Dr. Müller, 2008
Schürr, Andreas. Model Driven Engineering Languages and Systems: 12th International Conference ; Proceedings. Berlin: Springer, 2009
Sheard, Tim, Zine-el-abidine Benaissa, and Emir Pasalic. Domain Specific Language Construction Technology. Ft. Belvoir: Defense Technical Information Center, 2000
Soule, Paul. Autonomics Development: A Domain-Specific Aspect Language Approach. Basel: Birkhäuser, 2011.
Stahl, Thomas, and Markus Völter. Model-driven Software Development: Technology, Engineering, Management. Chichester, England: John Wiley, 2006.
Tāhā, Walid M. Domain Specific Languages: Ifip Tc 2 Working Conference ; Proceedings. Berlin: Springer, 2009.
Tairas, Robert, Marjan Mernik, and Jeffrey G. Gray. "Using Ontologies in the Domain Analysis of Domain-Specific Languages." Models in Software Engineering. (2009): 332-342.
Tratt, L. "Domain Specific Language Implementation Via Compile-Time Meta-Programming." Acm Transactions on Programming Languages and Systems. 30.6 (2008
Walter, T, and J Ebert. "Combining Dsls and Ontologies Using Metamodel Integration." (2009).
Widen, Tanya. Formal Language Design in the Context of Domain Engineering. Ft. Belvoir: Defense Technical Information Center, 2000.
Wijngaarden, A . Orthogonal Design and Description of a Formal Language. Amsterdam: Stichting Mathematisch Centrum, 1972
Wisse, Pieter. Metapattern: Context and Time in Information Models. Boston: Addison-Wesley, 2001
Wisse, Pieter. Semiosis &amp; Sign Exchange: Design for a Subjective Situationism, Including Conceptual Grounds of Business Information Modeling. Voorburg: Information Dynamics, 2002.
Yang, Zhihui. A Domain-Specific Modeling Approach for Component-Based Software Development. Muncie, Ind: Ball State University, 2009
Zschaler, S, D.S Kolovos, N Drivalos, R.F Paige, and A Rashid. "Domain-specific Metamodelling Languages for Software Language Engineering." (2010)

Biography

Kent Palmer has been a Systems Engineer and Software Engineer at a number of Aerospace companies. But he also has done research in Systems Theory, and Ontology over the years and formulated a theory called 'Special Systems Theory' that combines Reflexivity, Autopoiesis, and Dissipative Ordering of Structures in far from equilibrium conditions. He has just completed his Ph.D. in Systems Engineering at the University of South Australia, Defence and Systems Institute (DASI) with a dissertation on Emergent Design. Previous Ph.D. was in Sociology on Philosophy of Science at the London School of Economics, University of London with the title The Structure of Theoretical Systems in relation to Emergence.

---

[i] Dissertation of author at http://emergentdesign.net
[ii] See Wild Software Meta-systems by the author at http://works.bepress.com/kent_palmer
[iii] Pieter Wisse: http://www.informationdynamics.nl/pwisse/
[iv] GASM http://www.eecs.umich.edu/gasm/
[v] http://en.wikipedia.org/wiki/Unified_Modeling_Language
[vi] http://www.sysml.org/ or http://en.wikipedia.org/wiki/Systems_Modeling_Language
[vii] http://en.wikipedia.org/wiki/Domain-specific_language
[viii] http://en.wikipedia.org/wiki/KM3
[ix] http://en.wikipedia.org/wiki/INRIA
[x] GASM invented by http://research.microsoft.com/en-us/um/people/gurevich/ See also Egon Boerger http://www.di.unipi.it/~boerger/
[xi] Wisse, Pieter: Metapattern: context and time in information models (Addison-Wesley, 2001) http://www.informationdynamics.nl/knitbits/htm/primer.htm
[xii] Harold Ossher http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/o/Ossher:Harold.html
[xiii] http://systemengineering-lanomethodologies.com See also http://en.wikipedia.org/wiki/N2_chart
[xiv] See http://www.kfs.org/~jonathan/witt/tlph.html

[xv] http://en.wikipedia.org/wiki/Metamodeling
[xvi] Graham Priest: http://en.wikipedia.org/wiki/Graham_Priest
[xvii] http://en.wikipedia.org/wiki/Concept_of_Operations
[xviii] http://en.wikipedia.org/wiki/Source_lines_of_code
[xix] http://www.omg.org/mof/ http://en.wikipedia.org/wiki/Meta-Object_Facility
[xx] http://domaindrivendesign.org/ http://en.wikipedia.org/wiki/Domain-driven_design
[xxi] Euclid http://en.wikipedia.org/wiki/Euclid
[xxii] Peter Naur: See http://en.wikipedia.org/wiki/Peter_Naur
[xxiii] Charles Peirce: http://en.wikipedia.org/wiki/Charles_Sanders_Peirce
[xxiv] http://www.helsinki.fi/science/commens/dictionary.html
[xxv] Haack, Susan. Manifesto of a Passionate Moderate: Unfashionable Essays. Chicago: University of Chicago Press, 1998 page 55 Paraphrase "*precind, preciss precission precissive refers to the dissection of a hypothesis rather than an expression of determination freely or fully by the interpreter*" See 5.449 Collected Works of Peirce
[xxvi] http://www.eclipse.org/Xtext/
[xxvii] Terence Parr http://www.antlr.org/
[xxviii] Laurence Trat http://convergepl.org/
[xxix] Roman Knöll; Mira Mezini; Felix Wolff http://www.pi-programming.org/What.html
[xxx] As in the 'language reflection' feature See http://en.wikipedia.org/wiki/Reflection_(computer_science)
[xxxi] http://en.wikipedia.org/wiki/M_(programming_language)See also http://blogs.msdn.com/b/modelcitizen/
[xxxii] http://en.wikipedia.org/wiki/Oslo_(Microsoft)
[xxxiii] http://en.wikipedia.org/wiki/Object_Constraint_Language
[xxxiv] http://www.eclipse.org/modeling/emf/ See also http://www.eclipse.org/modeling/emft/?project=ecoretools