(4)

RADC-TR-89-183
In-House Report
October 1989

AD-A215 804

A MODEL INTEGRATION APPROACH TO ELECTRONIC COMBAT EFFECTIVENESS EVALUATION

Alex F. Sisti

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC TR-89-183 has been reviewed and is approved for publication.

APPROVED:

Thadeus J. Domurat

THADEUS J. DOMURAT
Chief, Signal Intelligence Division
Directorate of Intelligence and Reconnaissance

APPROVED:

Walter J. Senus

WALTER J. SENUS
Technical Director
Directorate of Intelligence and Reconnaissance

FOR THE COMMANDER:

Igor G. Plonisch

IGOR G. PLONISCH
Directorate of Plans and Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (IRAE) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS: N/A
2a. SECURITY CLASSIFICATION AUTHORITY: N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): RADC-TR-89-183	5. MONITORING ORGANIZATION REPORT NUMBER(S): N/A
6a. NAME OF PERFORMING ORGANIZATION: Rome Air Development Center; 6b. OFFICE SYMBOL (If applicable): IRAE	7a. NAME OF MONITORING ORGANIZATION: Rome Air Development Center (IRAE)
6c. ADDRESS (City, State, and ZIP Code): Griffiss AFB NY 13441-5700	7b. ADDRESS (City, State, and ZIP Code): Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Rome Air Development Center; 8b. OFFICE SYMBOL (If applicable): IRAE	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: N/A
8c. ADDRESS (City, State, and ZIP Code): Griffiss AFB NY 13441-5700	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO 62702F; PROJECT NO. 4594; TASK NO 15; WORK UNIT ACCESSION NO E1

11. TITLE (Include Security Classification)
A MODEL INTEGRATION APPROACH TO ELECTRONIC COMBAT EFFECTIVENESS EVALUATION

12. PERSONAL AUTHOR(S)
Alex F. Sisti

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
In-House	FROM Mar 89 TO Apr 89	October 1989	28

16. SUPPLEMENTARY NOTATION
N/A

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Modularity; Model Integration; Electronic Combat; Object-Oriented Design; Hierarchy of Models; Software Reuse; Model Management System (MMS)
15	04		
15	06.07		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report addresses the problems inherent in the modeling of large-scale and complex software systems in general, and specifically, how these problems have affected simulation systems designed to evaluate Electronic Combat Effectiveness in a combat scenario. Conceptual improvements and potential solutions are offered, leading to an in-depth discussion on a variety of disparate, yet related subject areas. An implementation of the Electronic Combat Effectiveness System is presented, based on these suggestions; and finally, recommendations are outlined as to future areas of research meriting increased investigation.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL: ALEX F. SISTI	22b. TELEPHONE (Include Area Code): (315) 330-4517	22c. OFFICE SYMBOL: RADC (IRAE)

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE

This report addresses the problems inherent in the modeling of large-scale and complex software systems in general, and specifically how those problems have affected simulation systems designed to evaluate Electronic Combat effectiveness in a combat scenario. Conceptual improvements and potential solutions are offered, leading to an in-depth discussion on a variety of disparate, yet related subject areas. An implementation of the Electronic Combat Effectiveness System is presented, based on these suggestions; and finally, recommendations are outlined as to future areas of research meriting increased investigation.

THE PROBLEM

The results of large-scale, monolithic battlefield simulation analyses have never really been totally accepted by management, and with good justification. A scenario of realistic proportions could conceivably involve the modeling of hundreds of thousands of entities, dynamically interacting among themselves, and reacting to other (simulated) activity in their environment. Even given the substantial hardware improvements in the form of larger, faster memories and exponential increases in processing power, it is still impossible to do an analysis at anything but a grossly aggregated level. It is becoming increasingly imperative that analyses of this sort pay more attention to the underlying details of the entities being modeled; especially since decisions based on these results could ultimately involve human life. As part of a panel discussion at the 1983 Winter Simulation Conference, panel chairman Kenneth Mussleman remarked "Aggregated measures are used to draw conclusions about system performance, while the detailed dynamics of the system go virtually unnoticed ... A decision based solely on summary performance could lead to unacceptable results ... It makes practical sense for us to learn how to properly amplify these details and to incorporate them into the evaluation process." One of the possible alternatives would be to accurately and completely model every entity in the scenario such that its associated details are therefore incorporated into the scenario. This approach is obviously discarded in light of the size and complexity of the simulation as well as cost, time, and resource constraints.

With the extremes (aggregate analysis based on coarsely represented dynamics versus completely detailed analyses) being rejected, what remains is: 1) modeling the complete system from a variety of different points of view, 2) modeling only selected items of interest, or 3) modeling the entire system, following a convention of multiple levels of representation, such that entities are modeled at varying levels of detail ranging from the top-level representation of the "essence" of the entity, to the lowest level, which would model the entity at the finest level of detail. This approach clearly has shown the most promise, and brings to bear a variety of technological, theoretical and practical aspects; including modularity, software reuse, object-oriented design, a hierarchy of models in a component library, a model management system for manipulating that library, and software engineering principles in general.

Modularity

Modularity is defined as breaking a software system into smaller and simpler parts or modules such that the modules together perform as the system. In the historical sense, a subroutine is functionally equivalent to a module which can be used repeatedly (and only when desired) in different places in the system. Breaking a large task into smaller, more tractable pieces is certainly not new; it is a time-honored method of increasing productivity in many manufacturing disciplines. It is therefore only natural that the earliest approach away from the traditional monolithic development of software systems was to modularize -- to decompose the system at the functional level -- and to group related functions together.

Leading the maturation of modularity is the thrust for improvements in the general area of software engineering principles, the foremost being the concepts of data abstraction, encapsulation and information hiding. Data abstraction refers to the process of hiding the implementation details of an object (e.g., program, data) from the users of that object; also called "information hiding" or "encapsulation". Abstract data types describe classes of objects as a function of their external properties as opposed to their specific computer representation. Deemphasizing a data type's representational details in this way can help eliminate problems of design changes, hardware changes and version compatibility. In other words, as the system undergoes a normal evolution, the actual implementation may be changed without affecting the users of the system. Examples of information likely to be hidden includes peculiarities of the underlying hardware, algorithms to implement a feature specified by the interface, representational details of data structures, and synchronization schemes.

Modular techniques formally began being embodied in programming languages in the mid-1970s with the development of Modula-2, with its "modules", and later, with Ada and its "packages". Still other, more conventional languages can now provide assistance in simulating data abstraction. These modular units contain separate sections for interface specifications and for implementation specifications, thereby supporting data abstraction. Making up the interface section would be information specifying type, data and procedures exported by the module, while the implementation section would contain the executable statements of the interface section, along with locally-used type, data and procedural declarations. Users wishing to invoke such a module to perform its function can access it through the interface section, but consistent with information hiding, cannot see (or affect) how that function has been implemented. The application of data abstraction and of a sister technology thrust, object-oriented design, will be discussed in a later section of this report.

As alluded to earlier, the reasons for modularity are many. A partial list follows:

1. Modularity facilitates writing correct programs. Smaller, more tractable modules, based on sound software engineering principles, are less susceptible to errors.
2. Modular code is easier to design and write. A software design engineer merely has to specify the functional essence of a module in a traditional top-down manner, rather than all of its internal and external details. Actual coding is then facilitated by this visibility of the design structure.
3. A modular approach to a system development allows many programmers to contribute to that development, in parallel. Domain-specific individual programmers can work on separate functional pieces, guided by the fact that interactions between parts of a system are rigidly restricted to the allowable interactions between those individual pieces.
4. Modularity facilitates easier maintenance as the system evolves. Since almost all large-scale software systems change as the requirements, methodologies or users change, it is essential that a system be easily reconfigurable and maintainable. Since modularity stresses the distinction between implementation changes and interface changes, modifications to the implementation may take place with confidence that inconsistencies will not be introduced to other parts of the system.
5. Modularity facilitates testing, verification and validation. Individual components can be "locally" tested, then fit into the system and re-tested. Furthermore, some applications have exploited modularity to the point of replacing software components with the hardware being simulated.
6. Modularity allows a component hierarchy to be exploited. The concept of a module hierarchy was alluded to earlier in the context of modeling components of a system at varying levels of detail. Bernard Zeigler, who has written many articles on the subject of hierarchical modular modeling [4,7,8,11,36] makes the point that "...models oriented to fundamentally the same objectives may be constructed at different aggregation levels due to tradeoffs in accuracy achievable versus complexity costs incurred." Hierarchical modeling will be discussed in much greater detail in later sections.
7. Lastly, and most important, modularity permits software reuse.

SOFTWARE REUSE

The process of combining and building up known software elements in different configurations (also called synthesis) has been likened to Gutenberg's concept of removable type. Gutenberg's contribution has historically endured the criticism of purists who argue that his printing press was not so much an invention, as a new application of existing

technologies at the time. To that, Douglas McMurtie responded in "The Book", "It does not at all minimize the importance of the invention ... to point out that the invention was the result of a process of synthesis or combination of known elements. For that power of the human mind which can visualize known and familiar facts in new relations, and their application to new ones -- the creative power of synthesis -- is one of the highest and most exceptional of mental faculties."

The idea of software reuse is certainly not new. Reuse and reworking has been practiced in one form or another since the 1950s; however, the landmark paper in this area is that of M.D. McIlroy's "Mass-Produced Software Components", published in 1968. In that paper, he envisioned and proposed a catalogue of software components from which "software parts could be assembled, much as done with mechanical and electrical components."

Software reuse is defined as the isolation, selection, maintenance and use of software components in the development and maintenance of a software project. Soundly based on the principles of modularity, it has been shown to improve productivity by using previously developed and tested components. Reusable components of a software system include design concepts, functional specifications, algorithms, code, documentation and even personnel. These components embody the various degrees of abstraction which pervade the process of classifying reuse items. Higher degrees of abstraction imply a greater likelihood for reuse. For example, specifications do not yet contain detailed representation details or implementation decisions, so the potential for reuse is greater, while it is very difficult to find pieces of code which can be used without some modifications.

Strictly speaking, software reuse must be distinguished from redesign or reworking. Reuse means using an entity in a different context than what was initially intended, and is also known as "black box" reuse. Redesign or reworking refers to the modification of an existing module before it is used in its new setting. This is known as "white box" reuse, and is by far the more common of the two.

Historically, the classical reusability technique has been to build libraries of routines (e.g., subroutines, functions, procedures), each of which is capable of implementing a well-defined operation. These routines are generally written in a common language for a specific machine, and are accessed by a linker as needed. Although this approach has met with some degree of success in numerical applications, there are some obvious problems which preclude using it to implement a generally applicable reuse system. Subroutines are too small, representation and implementation details have been filled in, and the glue (interface requirements) necessary to bring many subroutines together is too extensive to make general reuse feasible.

A second approach to reducing or eliminating the software development process takes the form of software generating tools. Software generation has been successfully applied in narrow, well specified domains (e.g., report generators, compiler-compilers, language-based editors), but shows little chance of being used outside these very specific domains. This is primarily because the nature of program generators is such that the application area needs to be very well-defined to achieve the desired level of efficiency.

The third and most promising approach for achieving reusability is based on a technique called object-oriented design. Under object-oriented design, the decomposition of a software system is not based on the functions it performs, but on the classes of objects the system manipulates. Object-oriented programming and languages support the notions of data abstraction, and encapsulation, as described above, and therefore exhibit the flexibility necessary to define and compose reusable components. Some of the more prevalent object-oriented languages include Simula, Smalltalk and some extensions of Pascal and Lisp. In addition, object-oriented principles and constructs are being implemented (or simulated) in other, more conventional languages; especially those which support data abstraction.

Assuming, as many practitioners have, that an object-oriented approach is the best way to describe a software reuse system, many other questions arise. How should existing systems be decomposed? What system fragments are candidates for reuse? How should these candidates be represented and stored? How should they be accessed? Once located, how should they be coupled with other candidate modules, such that together they perform as the system? In the section that follows, these questions are answered.

THE SOFTWARE REUSE SYSTEM

A conceptual software reuse system is made up of a component repository or library, from which the necessary components are retrieved and, if required, integrated with other components to address the functional requirements needed to solve a problem. The process is envisioned as follows:

The user, through some man-machine interface, presents the problem in a form which can be parsed to ascertain the functional requirements needed to solve the problem. The library management system would then search through the repository to find available candidate components such that, alone or in conjunction with other components, they can meet these functional requirements. If a component fully satisfies the requirements, nothing further is required of the reuse system; the component is simply executed as implemented. If some composition of components can solve the problem, they are coupled together as necessary by the reuse system, and are subsequently executed. However, more often than not, some

degree of modification of a component or components is necessary before reuse is possible. The very fact that modification of components is supported in a reus system is what makes software "soft". Biggerstaff and Richter [31] asserted that "The modifying process is the lifeblood of reusability. It changes the perception of a reusability system from a static library of rock-like building blocks to a living system of components that spawn, change and evolve new components with changing requirements with their environment."

This sequence, as described, implies many aspects of development that must be resolved up front, in order to progress from the current methods of processing; even in those systems that, at present, claim to be reusing software in some form or another. Design decisions have to be made regarding the nature of the component library. The reuse system designer has to decide on the functional areas that have to covered by the components. Once that decision is made, he can address the question of component availability -- does he have the necessary modules at his disposal? This often necessitates a survey of some magnitude, which will be needed at any rate for other development areas in building the reuse system (e.g., the library management system). In summary, a successful reuse system requires:

1) population of the component repository,
2) methods for accessing candidates in the repository,
3) methods for selecting candidates from the repository,
4) methods for modifying "close" candidate components and
5) methods for coupling candidate components

Populating the Component Repository

As a general rule, the first step in the development process should involve a survey of available modules/components that would satisfy the functional requirements of the system. That is, driven by the functional requirements of the problem to be solved, the designer should make a survey of the existing software, both internal to his organization and commercially available software packages. Of particular interest should be certain "reusability factors" that make some components more attractive (useful) than others. Examples of reusability factors are: the degree of language- and environment-independence, the extent to which the implementation is context-independent; the level of testing, validation, and verification (V&V) undergone by the component; the complexity and accuracy of the function being implemented, and the availability and quality of documentation.

Although a survey is certainly a good starting point, it is by no means the only method of identifying candidate components for inclusion. In many cases, the biggest step towards populating the component library involves a significant decomposition task -- a top-down pruning of the

functional structure of an existing software system to select the desired components. At this stage of the design, two major factors are kept in mind. In addition to the reusability factors just mentioned, the inherent hierarchical structure of the system may be exploited.

Zeigler [11], discussing the theoretical aspects of decomposition in a hierarchical sense based on varying degrees of detail, states "...specification of design in levels in a hierarchical manner [implies] the first level, and thus the most abstract level, is defined by the behavioral description of the system. Next levels are defined by decomposing the system into subsystems (modules, components) and applying decompositions to such subsystems until the resulting components are judged not to require further decomposition...Therefore, the structure of the specification is a hierarchy where leaf nodes are atomic models (cannot be decomposed any further)." Hierarchical decomposition/representation is fairly extensively treated in the literature; by Zeigler and others. Zeigler's conceptual design is covered later in this paper.

Accessing the Component Library

Once identified and "accepted" for inclusion, the component needs to be stored in the component library, in a manner which facilitates straightforward accessing, comparison to the input requirements, and ultimately, retrieval. One outstanding approach taken by Burton and others in "The Reusable Software Library" [25] involves the insertion of a "reuse comment" section into each component, which can be automatically scanned by the library management system. Each reuse comment is treated as an attribute of the component itself; e.g. the comment labeled 'Overview' contains a description of the component's function. Components are first scanned for possible entry in the library, addressing the factors of reusability and the analysis of interest. Once components are entered into the library, the problem of searching for components to match the analysis requirements is reduced to one of scanning these reuse comments and extracting the pertinent components for coupling and/or execution.

Selecting Candidates

As components are scanned in the conceptual reuse system, their applicability to the problem is assessed. This is basically a matching problem, which can be handled in a variety of ways; none very easy. Given the user's problem (experiment, area of analysis, etc.), the library management system has to search through the available components to see if any one component, or a mix of components, can solve the problem. One approach might take the form of a table lookup, involving a "requirements versus components" matrix; an implementation which again requires considerable up-front effort to populate these matrices. Another more elegant implementation incorporates a "closeness metric" as a way of

choosing and ranking likely candidates. More often than not, several candidates exist, each satisfying some of the requirements. These measures of closeness are useful in selecting and ranking the available candidates based on how well (how closely) they matched the requirements of the system. A third method of candidate selection is slightly more futuristic, and is based on an Expert System-based implementation. As part of the library management system, there could exist a "decision rule base", which incorporates component selection knowledge, including explanations as to how and why some candidates were selected over others.

Modifying "Close" Components

As mentioned earlier in this paper, the ability of the reuse system to adapt existing components to the evolving requirements of the project is (at present) its most attractive feature. The distinction between implementations of the present and those of the future is intentional -- a reuse system being developed today, from existing components or products of a decomposition process is seriously constrained from the start. A modifying process is essential in such a system, because with very few exceptions, existing components are not amenable to reuse without modifications of some sort. It is hoped that in development efforts in the future, software components will be designed with reuse in mind from the outset, encompassing the recommended software engineering principles discussed earlier. Until then, however, the conceptual reuse system and the associated library management system should be capable of modifying "close" components to better meet the specifications of the user's problem. Today, this is largely a manual process, as few tools exist to help modify components. Among other things, this process could include prompting the user to interactively supply, for example, a missing data item or unit conversion. Of course, a well-designed library management system could eventually resolve these differences, even going as far as accessing a database or suggesting the execution of other components to supply the missing information.

Component Coupling

The most difficult aspect of the conceptual reuse system deals with coupling, or integrating, existing components together in some manner, to replicate the desired functionality of the system. The benefits of component coupling were discussed in an earlier section of this paper. Therefore, this section will address the implementation aspects of component coupling.

Whereas system decomposition is viewed as a classic application of top-down design, component synthesis takes a bottom-up approach. Candidate components meeting the input specifications are put together like building blocks to construct a software system capable of solving the problem. As discussed earlier, these individual components should satisfy

the software engineering principles of abstraction, encapsulation and information hiding to reduce the amount of (usually manual) modification needed to integrate them. Software components written in languages which support those features are obviously most desirable from an integration point of view, but that is not to say that software (subroutines, functions etc.) written in a conventional language cannot be used. Again, one of the major design considerations of a reuse system involves possible workarounds because of a desire to retain an expensive or "key" piece of software.

There are two fundamental ways components can be integrated, depending on the degree of interaction required and inter- and intra-dependence of individual components. In the first and simplest case, software modules are run separately and sequentially, using outputs of one component as inputs to the next component to be executed. This method, alternately called chaining or the UNIX pipeline method, is the most straightforward method of integration, and is easily implemented; although some interim transformations or analysis may be required. The second integration method is more complex (often impossible) and is employed when the components are expected to interact with each other. The composition principle used in this case is based on inheritance and message-passing. As expected, the components in a reuse library using this integration method should preferably be those which embrace the policies of object-oriented design, as their interface details are well specified and they can be bound with other components without the internal (implementation) details of the components being known. Object-oriented programming also supports the concepts of object classes and the message-passing between objects. The notion of inheritance is applied in passing messages between classes and subclasses (parents and children). When a new object is sub-classed from a more generic parent class, capabilities common to both are implemented. In addition to being able to process any message that the parent class can (by passing off to the more generic parent), the sub-class can locally process messages which are specific to itself. Again, Ada's "generic packages" and Modula's "modules" meet these objectives. Another useful construct in Ada and some other object-oriented languages which helps facilitate integration is known as "overloading". Overloading allows more than one meaning to be attached to a name. To use an example from the literature, giving the name "Search" to all associated search procedures enables the user (or the library management system) to always invoke a search operation in the same manner, regardless of the implementation chosen, or the data types. Still another construct being advocated is that of semantic binding. This applies to the flexibility needed when referencing across different domains. The calling component must refer to items it expects in its context, and since it cannot know the items' names in advance, it should be able to refer to them semantically.

One can perhaps begin to see the intricacies involved, and the enormous representational decisions required in developing a reuse system. Reuse systems and their associated tools are costly in terms of time, money and personnel, and even then are often dismissed as unfeasible due to lack of reliable, reusable components. Contractors are hesitant to build software that is too reusable for fear that there may be no "next job" for him. Even the "not invented here" syndrome causes resentment among management and system users. Most importantly, reuse systems have proven to be very application-dependent, with some applications providing a more mature technology base on which to build such a system. One such application area is in support of Modeling and Simulation.

REUSE AND MODULARITY APPLIED TO MODELING AND SIMULATION

Reese and Wyatt [1] speaking at the 1987 Winter Simulation Conference stated "The issues of reusability ... apply to all types of software, including simulation software ... Adoption of reuse philosophy and the subsequent creation of a reuse library by management is expected to improve the simulation software development process and increase the credibility of simulation results." In addition to the standard components mentioned in earlier sections of this paper, there are some support functions which are fairly specific to modeling and simulation, and are required by nearly all discrete simulation applications. Examples of such functions are: time management; queue management; random number generation; data I/O; input sequence front-ends; debug routines; validation and verification tools; graphical/statistical analysis tools, and animation tools. Obviously, the functions of individual model components are also amenable to reuse technology. In the sections that follow, the discussions on reuse are instantiated to the field of modeling and simulation technology. As part of this instantiation, some of the previously used terms will be changed for clarity. That is, components will be called models; the component library will be designated the Model Base; the library management system will be referred to as the Model Management System (MMS), and component coupling will, in general, be known as model integration.

Zeigler's Hierarchical, Modular Modeling

Arguably the most prolific of the authors and researchers in the domain of model integration is Bernard P. Zeigler, who first referred to the concept in a 1976 book entitled "Theory of Modeling and Simulation." In that book, he quietly introduced the idea of decomposition of existing models in a hierarchical manner, corresponding to the levels of functional detail. He discusses the process of aggregating the details of what he calls base models into "lumped" models. A lumped model includes the functional coverage of one or more detailed component (base) models, modeled with less detail. This involves a simplification process in which the description of the base model is modified in one of the following ways: 1)

one or more of the descriptive variables are dropped and their effect accounted for by probabilistic methods, or 2) the range set of one or more of the descriptive variables is coarsened, or 3) similar components are grouped together and their descriptive variables are aggregated. Finally, he pursues a mathematical approach to valid simplification, based on "structure morphisms" at various levels of specification.

In later works, he refined his ideas, and changed the term "lumped" model to coupled model. Most of his contributions to the literature now revolve on his hierarchical, modular composition techniques or actual implementations of his concepts. He says [8] "Considering a real system as a black box, there is a hierarchy of levels at which its models may be constructed ranging from purely behavioral, in which the model claims to represent only the observed input/output behavior of the system, up to the strongly structural, in which much is claimed about the structure of the system. Simulation models are usually placed at the higher levels of structure and they embody many supposed mechansims to generate the behavior of interest."

Central to his hierarchical scheme is the composition tree, which describes how components are coupled together to form a composite model. In his words (and referring to Figures 1 and 2), "Suppose that we have models A and B in the model base. If these model descriptions are in the proper form, then we can create a new model by specifying how the input and output ports of A and B are to be connected to each other and to external ports, an operation called coupling. The resulting model, AB, called a coupled model is once again in modular form ... modularity, as used here, means the description of a model in such a way that it has recognized input and output ports through which all interaction with the external world is mediated. Once in the model base, AB can itself be employed to construct yet larger models in the same manner used with A and B. This property, called closure under coupling, enables hierarchical construction of models." Noting the dual relationship of system decomposition and model sysnthesis, he remarks that the coupling of two atomic models A and B is associated with the decomposition of the composite model AB into components A and B.

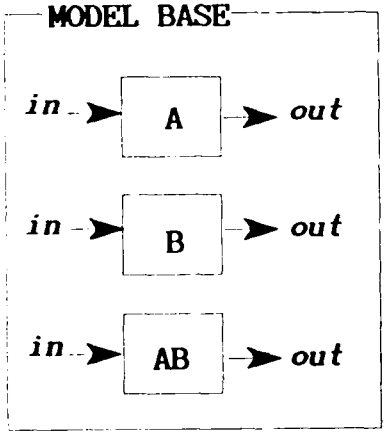
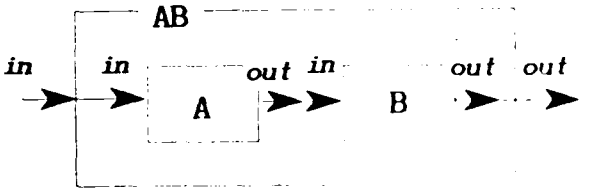


Fig 1. Model Base



COUPLING:

external input: AB.in -> A.in

external output: B.out -> AB.out

internal: A.out -> B.in

Fig 2. Model Coupling

In Zeigler's scheme, which has already been applied to a variety of disciplines, there are three basic parts to the description of an atomic model: 1) the input/output specification, explicitly describing the input and output ports and the ranges their associated variables can assume, 2) the state and auxiliary variables and their ranges (the static structure) and 3) the external and internal transition specification (the dynamic structure). The descriptions for coupled models differs slightly, in that information pertaining to the coupling process is also included. Separate files containing interface specifics are maintained for each component and facilitate the coupling of selected models.

There are three facets to Zeigler's coupling scheme. First, there is a file which contains information relating the input ports of the composite model to the input ports of the components (called external input coupling). Next, external output coupling tells how the output ports of the component model are identified with the output ports of the components. Finally, internal coupling information is maintained, telling how the output ports of the components are connected to input ports of other components; in other words, it describes how the components inside a composite model are interconnected. Following the principles of abstraction and object-oriented design, all interaction with the environment is mediated through these input and output ports, regardless of the internal implementations of the models. Furthermore, the sending of external events from the output port of one component to the input port of another component can be likened to message passing; another composition technique supported by object-oriented design.

This multifaceted, hierarchical modular modeling concept has been successfully applied in other disciplines; and each time, is improved upon to some degree. The lion's share of his research has been in the representational details of the model base (the component library) and to a limited extent, maintenance of that library. To fully instantiate the idea of a reuse library management system to the field of Modeling and Simulation, it is once again necessary to borrow from one of the kindred fields of software science; that being Decision Support Systems research.

THE MODEL MANAGEMENT SYSTEM (MMS)

Model Management has recently been extensively studied and applied by Benn Konsynski and others [37-44] in relation to its function in a Decision Support System. Essentially, it is an instantiation of the library management system discussed in earlier sections, providing for the creation, storage, manipulation and accessing of models in a model base. In other words, a Model Management System (MMS) is to the Model Base what a DBMS is to a database. There are three basic components which make up the MMS: The Decision Component, the Model Component and the Data Component.

The Decision Component provides the user with the ability to describe, analyze and store decisions. It is comprised of a Decision Manipulation component and a Decision Storage component. An example of what might be found in the Decision Manipulation component would be solution rules or model selection rules. The Decision Storage component contains, for example, the rules related to completeness/consistency checks. The Model Component is made up of a Model Manipulation component and a Model Storage component. Examples of the types of functions found in the Model Manipulation component are model classification rules and data selection rules. An example of what would be found in the Model Storage component would be rules for checking the consistency/integrity of the models. Finally, the Data Component is basically a DBMS.

The most elucidating information on the workings of an MMS comes from knowing the interactions among the three components of the MMS, and between the MMS and the User Interface to the MMS.

Interactions involving the Decision Component:

The Decision Component accesses the Model Component to retrieve, sequence and control the models needed to solve a specific problem. It checks the consistency, integrity and completeness of the Model Base and the database for solving a specific decision problem. Finally, it queries the user through the User Interface if the data or models needed to solve the problem are inconsistent or incomplete.

Interactions involving the Model Component:

The Model Component accesses the Data Component to retrieve, sequence and control the data needed for implementing a specific model.

Interactions involving the User Interface:

At any time, the user may interactively input data or models that are not stored in the system, through the User Interface. Also, the user can directly access the Model Base or the database for model or data management and analysis that is not specifically related to a particular decision problem.

The Model Management System is the cornerstone of a software reuse system, and can be extremely complex in design. Much of the literature in this area confines itself to presenting knowledge representation schemes for implementing a conceptual MMS, but little has been formally done as far as actually building one. Most often, a manual, brute force approach is taken for Model Base population and manipulation. Such is the case for the modeling system described in the next section.

ELECTRONIC COMBAT EFFECTIVENESS ANALYSIS: The Concepts are Applied

The specific application of software reuse and model integration of interest to this paper involves conducting simulation studies to evaluate the effects of Electronic Combat (EC) in support of mission planning in battlefield scenarios. In addition to the problems which are universally common to software reuse in general, some domain-specific issues are introduced in studies of this sort. In contrast to earlier building block applications of software reuse which, in general, use decomposition schemes based primarily on the implicit functional hierarchy of the system, this study includes another aspect which can be hierarchically described; that being the degree of analysis possible at each level. In a 1979 Air Force study, LtCol (then Major) Glen Harris addressed this new aspect in regards to analyzing force effectivess, and introduced the concept of a "validated analytical hierarchy of models", stating "Neither a highly detailed approach nor a broad aggregate modeling approach by itself is adequate to analyze the complex battlefield. Unless both approaches are used and carefully integrated, the results obtained will not provide the insight required to determine why one ensemble of systems should be preferred over another. An integrated approach must be designed to answer several levels of questions as to the causal relationships involved..." Accordingly, various working groups under the Department of Defense defined four levels of Electroinc Combat analysis, as follows:

Level I: System/Engineering Analysis. The analysis at this level primarily deals with individual systems or components; e.g., jammers, sensors, transmitters, antennas, etc. The objective is to measure the required and/or achieved engineering-level data for Electronic Warfare systems and their interactive effects with target systems [46]. The analysis at this level (and therefore any Measure of Effectiveness associated with this level) is limited to the effects of, for example, a single jamming component against a single target threat.

Level II: Platform Effects Analysis. At this level, the evaluation focuses on the component being associated with a platform; e.g., a radar jammer installed on an aircraft. The effectiveness of the installed system is then evaluated in the context of a one-on-one or few-on-few analysis.

Level III: Mission Effectiveness Analysis. Analysis at this level assesses the contribution of Electronic Combat to a combat mission environment, including other aspects such as Command and Control, time-sensitive manuevers, and a defined enemy posture.

Level IV: Force Effectiveness Analysis. This encompasses all the activity associated with operations in the context of joint Air Force/Army/Navy campaigns against an enemy combined

arms force, towards evaluating the contribution of Electronic Combat support in such a campaign.

Under this hierarchical scheme, a distinction is made between vertical integration and horizontal integration. Vertical integration refers to the ability to pass data output of a model at one level of the hierarchy to the input of a higher (or lower) level model. This is in consonance with the concept of the UNIX pipeline method of integration discussed earlier. Another method of vertical integration is effected by using lower level models as modules in higher level models. For example, a Level I standard propogation model could be used in a Level II Surface-to-Air weapons model, which could in turn be used in a Level III mission effectiveness model. Vertical integration is important because it provides a validated audit trail of higher-level results to "hard" engineering data, range data, hybrid simulation outputs and/or flight test data. The credibility of most of the upper-level modeling rests on the ability to vertically integrate.

Horizontal integration (federation) of data refers to the accessing (by models at all levels) of a master input/runtime database for data that is global in nature. This concept eliminates the problem of needing multiple databases for multiple models. When intelligence and other situational (state) changes require updating of data, they are changed in one place only, with new values propogating to all models that need to reflect those updates; analogous to the blackboard approach followed in some disciplines of Artificial Intelligence.

The Problem

The specific problem which is the subject of this paper deals with the lack of fidelity of an existing mission planning tool. Although soundly based on phenomenological and physical properties of barrage jamming (radiated power against radiated power), the tool lacked the required depth, as far as simulating existing Electronic Combat assets to the detail needed for real-life decision making. As in other disciplines, when users required this additional detail, a decision had to be made as to its implementation: rebuild or reuse. For the many reasons described earlier, an integration approach was deemed 1) the most attractive from a cost/time perspective, 2) the most intriguing from a research and development standpoint, but 3) certainly the most difficult (and possibly unattainable), from a realistic point of view.

A Solution

Complexities notwithstanding, an integration approach was pursued. First, existing model repositories were surveyed, as to their compliance with the well-designed criteria of the survey. For instance, each candidate model was compared against such features as model availabilty, degree of validation, modeling methodology (e.g. stochastic event

scheduling versus deterministic scripting), underlying assumptions, Measures of Effectiveness, host software/hardware, dependencies on other models/databases, processing modes (interactive versus batch), availability and currency of documentation, and others too numerous to mention. Finally, three candidate models were chosen, representative of a specific radar jammer, a specific communications jammer and an aircraft dedicated to suppressing enemy air defense systems. These models were obtained from their respective owning/maintaining agencies, and were microscopically studied to ascertain and illucidate the necessary integration properties; such as input/output characteristics, units, etc. It should be stressed here that this was a very time- and manpower-intensive undertaking, necessitated in general because of the absence of object-oriented principles in any of the chosen models.

Once modified (minimally, so as not to violate the requirement of maintaining each model's standalone status), the three models were configured to run synchronously under the existing simulation executive. Together, the integrated system is called the Electronic Combat Effectiveness System (ECES), and runs on a VAX 11/780. At startup, the three models were informed of the initial conditions of the scenario (e.g., each's own flight path, the perceived threat laydown, geographical/terrain data, etc.), which were, in general, required inputs of each to begin with. In addition, each model contained (or read) its own local data, necessary for standalone execution.

As the ECES model (the aggregate model) is run, messages are sent to the individual (component) models to update positions, announce threat movements and signal activity, and so on. In return, each model passes messages such as flight path changes, jamming noise figures (if requested), or other information pertinent to the mission. The aggregate model serves a dual purpose: it is the orchestrator of the event script during the scenario (updating the information common to all the models), as well as dynamically displaying all scenario activities (common to all, or as specifically reported by each model) on a graphics terminal.

The Electronic Combat Effectiveness System now runs as it did before its decomposition; the difference with the current system is that validated jammer characteristics are now explicitly modeled, and are inherited as needed. Another major plus is that the synergistic effects of the Electronic Combat assets (taken pairwise or in total) can now be determined. This offers an obvious improvement over the independent execution of the three component models in standalone mode.

During the execution of the system, statistics are collected for ultimate Measures of Effectivess (MOEs) calculation, corresponding to the four levels of the analysis hierarchy desribed earlier. For example, in determining mission-level (Level III) effectiveness, a factor called "per

cent neutralization" is computed, comparing the ability of (for example) a Target Tracking Radar to successfully track an incoming strike aircraft; both in the presence and absence of jamming support. The actual MOEs and overall results of the asset tradeoffs, while interesting, are beyond the scope of this paper. What is of greater interest is the fact that, despite the labor-intensive nature of the integration effort, software reuse was possible, and actually (in this application) facilitated a "total-is-greater-than-the-sum-of-its-parts" system which was not attainable through independent execution of the component models.

One rather glaring omission in the system was the lack of a Model Management System, per se. Since the library only consisted of the three candidate models, model selection was trivial. Obviously, now that the concept and a chosen integration methodology have been proved feasible (and useful), the next step is to obtain, build or acquire additional modular components for populating the component library. Armed now with the lessons learned during this integration effort, more consideration will likely be given towards assuring that any modules/models selected for inclusion adhere to the principles of data abstraction, encapsulation, and so on. These lessons learned will not only help in the next integration effort we undertake, but will also no doubt help define future directions in research in the field.

FUTURE RESEARCH DIRECTIONS

In the future (and starting now), research should be done in the following areas: model/software development, Model Management Systems and standardization. As mentioned above, stricter adherence to modern software engineering principles is a must. It appears that those development efforts that are founded on object-oriented design -- Ada in particular -- represent a good start towards expanding the technology base. In addition, new model structures may be developed which specifically address the issues of software reusablility and integration. Model Management Systems and their associated tools must be developed to facilitate the selection and synthesis processes in configuring a "solution model" for a given problem. Expert System techniques could be brought to bear in nearly all phases of development of the Model Management System; especially for choosing and coupling certain mixes of models -- even offering explanations as to why they were chosen over others. The User Interface portion of the MMS suggests a fertile area for improvement. Just as window-based interfaces and mouseable items have progressed rapidly over a short time, other advances are likely in the areas of natural language interfaces, touch- and voice-sensitive input and visual languages [21]. Other research areas which could be applied to Model Management System development include database development, distributed and parallel processing, library representation schemes; and the list literally goes on and on. Underlying everything, however,

is one common thread: standardization. Having been successfully applied in nearly every known discipline, standardization also holds great promise in the field of software reuse. As models/modules are developed and eventually evolve into sucessfully reusable components, they could be made available as standard component models. Standard components could then be coupled to form standard aggregate models, if needed for some analyses. Eventually, it is conceivable that packaged libraries of optional standard reusable components could be offered, much as libraries of mathematical routines are available as part of compiler packages. A less obvious area requiring standardization deals with the problem of integration. Compatibility standards should be investigated, against which newly written models would be required to comply. Standard "closeness metrics" should be defined, to rate candidate components equally. Finally, some thought should be given to even standardizing units for often-used parameters; possibly even those parameter names. Again, this list is only meant to show a few of the many areas that are amenable to standardization.

CONCLUSION

Despite documented prophesies to the contrary, software reuse and model integration will continue to grow. As expected improvements in programming practices and standardization material ze, reuse system development will migrate from this awkward period of "working with the givens", towards the inception of standard models, interfaces, and perhaps even packaged standard libraries. This next generation of software system development will not be without cost. There will be great outlays and sacrifices in all areas; not the least of which will be obtaining management support. But no matter the cost, no matter the effort, no matter the level of resistance encountered, this technology area should be vigorously pursued, as were many other unlikely, yet promising areas, which are now standard practices. Simply put: for large-scale software system development, there is no realistic alternative.

BIBLIOGRAPHY

1. R. Reese, D. L. Wyatt, "Software Reuse and Simulation", Proceedings of the 1987 Winter Simulation Conference

2. G. C. Vansteenkiste, "New Challenges in System Simulation", Proceedings of the 1985 Summer Computer Simulation Conference

3. K. J. Murray, S. V. Sheppard, "Automatic Model Synthesis: Using Automatic Programming and Expert Systems Techniques Toward Simulation Modeling", Proceedings of the 1987 Winter Simulation Conference

4. B. P. Zeigler, T. G. Kim, "The DEVS Formalism: Hierarchical, Modular Systems Specification in an Object Oriented Framework", Proceedings of the 1987 Winter Simulation Conference

5. D. Kostelski, J. Buzacott, K. McKay, X. Liu, "Development and Validation of a System Macro Model Using Isolated Micro Models", Proceedings of the 1987 Winter Simulation Conference

6. R. G. Sargent, "An Overview of Verification and Validation of Simulation Models", The Proceedings of the 1987 Winter Simulation Conference

7. B. P. Zeigler, "Hierarchical Modular Modeling/Knowledge Representation", Proceedings of the 1987 Winter Simulation Conference

8. B. P. Zeigler, T. I. Oren, "Multifacted, Multiparadigm Modelling Perspectives: Tools for the 90's", Proceedings of the 1987 Winter Simulation Conference

9. R. G. Sargent, "Joining Existing Simulation Programs", Proceedings of the 1987 Winter Simulation Conference

10. A. I. Concepcion, S. J. Schon, "SAM - A Computer Aided Design Tool for Specifying and Analyzing Modular, Hierarchical Systems", Proceedings of the 1987 Winter Simulation Conference

11. J. W. Rozenblit, S. Sevinc, B. P. Zeigler, "Knowledge-Based Design of LANs Using System Entity Structure Concepts", Proceedings of the 1987 Winter Simulation Conference

12. S. A. Shoaf, "A Modular Approach to the Simulation of Manufacturing Processes", Proceedings of the 1983 Winter Simulation Conference

13. K. J. Musselman, et al, "Practitioners' Views on Simulation", Panel Discussion from the Proceedings of the 1983 Winter Simulation Conference

14. W. T. Jones, B. J. Jones, "Computer Simulation Using Hierarchical Models", Proceedings of the 6th Pittsburgh Conference, 1975

15. B. R. Konsynski, J. F. Nunamaker, "A Generalized Model for Computer-Aided Process Organization in Design of Information Systems", Proceedings of the 6th Pittsburgh Conference, 1975

16. D. W. Balmer, "Modelling Styles and Their Support in the CASM Environment", Proceedings of the 1987 Winter Simulation Conference

17. R. Prieto-Diaz, P. Freeman, "Classifying Software for Reusability", IEEE Software, Jan 1987, p6

18. R. F. Kamel, "Effect of Modularity on System Evolution", IEEE Software, Jan 1987, p 48

19. A. Reilly, "Roots of Reuse", IEEE Software, Jan 1987, p 4

20. B. D. Shriver, "Reuse Revisited", IEEE Software, Jan 1987, p 5

21. S. Chang, "Visual Languages: A Tutorial and Survey", IEEE Software, Jan 1987, p 29

22. W. Tracz, "Reusability Comes of Age", IEEE Software, Jul 1987, p6

23. P. G. Bassett, "Frame-Based Software Engineering", IEEE Software, Jul 1987, p 9

24. G. E. Kaiser, D. Garlan, "Melding Software Systems from Reusable Building Blocks", IEEE Software, Jul 1987, p 17

25. B. A. Burton et al, "The Reusable Software Library", IEEE Software, Jul 1987, p 25

26. M. Lenz, H. A. Schmid, P. F. Wolf, "Software Reuse Through Building Blocks", IEEE Software, Jul 1987, p 34

27. A. Gargaro, T. L. Pappas, "Reusability Issues and ADA", IEEE Software, Jul 1987, p 43

28. S. N. Woodfield, D. W. Embley, D. T. Scott, "Can Programmers Reuse Software?", IEEE Software, Jul 1987, p 52

29. G. Fischer, "Cognitive View of Reuse and Redesign", IEEE Software, Jul 1987, p 60

30. R. Conn, "ADA Software Repository", IEEE Software, Jul 1987, p105

31. T. Biggerstaff, C. Richter, "Reusability Framework, Assessment, and Directions", IEEE Software, Mar 1987, p 41

32. B. Meyer, "Reusability: The Case for Object-Oriented Design", IEEE Software, Mar 1987, p 50

33. K. W. Miller, L. J. Morell, F. Stevens, "Adding Data Abstraction to Fortran Software", IEEE Software, Nov 1988, p50

34. G. Gruman, "Early Reuse Practice Lives Up To Its Promise", IEEE Software, Nov 1988, p 87

35. J. R. Emshoff, R. L. Sisson, "Design and Use of Simulation Models"

36. B. P. Zeigler, "Theory of Modeling and Simulation"

37. Dolk, B. Konsynski, "Knowledge Representation for Model Management Systems", IEEE Transactions on Software Engineering, Vol SE-10, No. 6, Nov 1984

38. O. I. Truncer, "Concepts and Criteria to Assess Acceptability of Simulation Studies: A Frame of Reference", Communications of the ACM, Vol 24, No. 4, Apr 1981

39. Klein, Konsynski, Beck, "A Linear Representation for Model Management in a DSS", Journal of Management Information Systems, Vol II, No. 2, Fall 1985

40. McIntyre, Konsynski, Nunamaker, Jr., "Automating Planning Environments: Knowledge Integration and Model Scripting", Journal of Management Information Systems, Vol II, No. 4, Spring 1986

41. Liang, Ting-Peng, "Integrating Model Management with Data Management in DSS", Decision Support Systems 1 (1985), p221

42. Applegate, Konsynski, Nunamaker, Jr., "Model Management Systems: Design for Decision Support", Decision Support Systems 2 (1986), p81

43. Konsynski, Sprague, "Future Research Directions in Model Management", Decision Support Systems 2 (1986), p 103

44. Fedorowicz, Williams, "Representing Modeling Knowledge in an Intelligent Decision Support System", Decision Support Systems 2 (1986), p3

45. G. L. Harris, "Computer Models, Laboratory Simulators and Test Ranges: Meeting the Challenge of Estimating Tactical Force Effectiveness in the 1980's", 1979

46. G. R. Dougherty, "On What Basis, EW?", Journal of Electronic Defense, Oct 1984

47. A. F. Sisti, et al, "Electronic Combat Development and Demonstration Component", RADC-TM-86-9, Aug 1986

48. A. F. Sisti, et al, "Automated Intelligence Decision Aids", RADC-TR-87-17, Feb 1987

49. M. Ringler and G. Brown, "Electronic Combat Effectiveness Study" Final Technical Report, Jul 1988

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic reliability/maintainability and compatibility.