Simulation-Based Acquisition: Architecture and Implementation, Part 2

Will Hollier
Engen Institute
will@engen.org

Paul Beckett
School of Electrical and Computer Engineering, RMIT University
PBeckett@rmit.edu.au

Abstract. The requirements to implement the Simulation Based Acquisition vision as analyzed in Part 1 are reviewed. Domain Oriented Design Environments are characterized and an implementation suitable for SBA presented. Three stages of system integration and interoperability based on semantics, program generation & renovation and strategic programming are presented.

1. INTRODUCTION
The goal of Simulation Based Acquisition (SBA) is the accessibility and management of all organizational processes, both technical and managerial, using simulation. [1],[2].

In Part 1, we examined the objectives of SBA, its architecture and state of development and found that there was some developed technology and a need for supporting technology. However the key technical sub-system for Requirement Driven Design was not only ill defined but could not be defined with current or proposed systems engineering methodologies.

Having developed a design environment for conceptual and systems level design, capable of generating detailed engineering specifications which are executable (ie simulate-able), it is then possible to proceed to implement the mid-term integration goal of SBA. This mid-term integration requires plug and play simulations and semantically defined data.

A definition of the high level of integration that is the ultimate goal of both the SBA Initiative and NASA’s Integrated Synthesis Environment Initiative was being sought by the Pathfinder experimental programs. The analysis in Part 1 concluded that process transformation would be required to achieve the desired functionality which is beyond the scope of Pathfinder.

In Part 2, particular software engineering methodologies will be described which will allow implementation of a requirements driven design environment – the essence of SBA.

Secondly, knowledge based systems generation and transformation methods capable of realizing the high level of process integration and optimization sought by the SBA Initiative are described.

2. REQUIREMENTS FOR SBA
Part 1 identified the following actions to implement SBA

A. Implement Developed Technology -
1. Need Business Simulation for Capability design
2. Implement Business M&S via Process Modelling
   (a). Process Modelling (IDEF standards & Tools)
   (b). Documentation (XML CALS and IETM’s)
3. Implement Engineering CAD and PDM
   (a). STEP Standard & Tools for CAD interchange
   (b). Use Product Data Models (PDM) tools
   (c). Possible to use Model & Simulation based CAD
4. Implement small supplier network with legacy data processing for CALS, STEP, PDM and IETMs.

B. Develop Design Environment for early lifecycle using
5. Defined Requirements Driven Development Process
6. Develop Requirements Driven Development Tools

C. Software Transformation & Simulation

Intergation

7. Integration via semantically unified data architecture
8. Integration via process transformation & generation
9. Integrated AI including constraint solving

The developed technology is available as commercial software and although implementation is non trivial it can be achieved by a competent professional utilizing standard software technology and tools and will not be discussed further.

Sections 3 to 5 describe the design and implementation of Domain Oriented Design Environments similar to those used in software and electronics design systems with capabilities to support defense development.

Sections 7 to 9 describe software engineering methods of increasing capability which can be used to realize both the near-term and long-term integration, interoperability and re-use objectives of the SBA initiative. However these techniques can also enhance SBA with capabilities such as automatic documentation, proof of correctness and optimization.

**Figure 2: Design Environment (from [4])**

Section 10 outlines an implementation program that employs the relationship between these technologies to incrementally establish all of the above capabilities.

3. DESIGN ENVIRONMENTS

Design Environments [4] have developed independently in several disciplines –
Programming Tools & Software Generation [5]
Electronic, IC & Microsystem design [4],[7]
Expert Systems Knowledge Acquisition [6]
Defense Development Domain Expertise [6]
Program Transformation & Synthesis [18],[19],[21]

Design Environments support design, analysis and testing of designs and not just design generation. Design Environments usually provide some automation of design or design generation. Design environments should also allow the capture and retention of partial knowledge such as concepts and characterizations.

Domain Oriented Design Environments can integrate multiple aspects of a system and/or multiple subsystem design via multiple graphic domains with corresponding graphic languages. **Figure 3: Electronic Design Simulation Stack (from [4])**

Design Environments can not only represent levels of detail and different subsystem technologies [4] but are frequently used to represent the various stages in design evolution. **Figure 4: Design Refinement (from [7])**

It is this ability of design environments to deal with multiple forms of design and multiple representations of a design from the concept stage through to detailed design and manufacturing information that make design environments suitable for simulation support of defense development.

The SBA Initiative calls for interoperability between Virtual Prototypes, Simulation Based Design and STEP (Simulate, Test, Evaluate Program) which encompasses the Simulate, Emulate and Stimulate progression. As well as design environments the
authors are researching the integration of Virtual Reality and Tele-Presence to produce a ‘mixed reality’ Development Environment.

Initially a Capability Statement can be ‘flown’ in VR by simulating its missions in the geo-spatial environment (with culture including defense facilities and their logistic rolls) that models the ‘AS IS’ defense organization. Technology Opportunities can be evaluated by adding them to the mission profiles, via SBD detail engineering can be added to the growing platform model, then via hardware-in-the-loop production information is added, and finally field testing accomplished via Tele-Presence.

3.1 Design Environment Architecture

The many sources from which design environments have originated have resulted in design environments being implemented via numerous software methods and architectures. However some general principles apply to the architecture of design environments and the software methods employed directly relate to the final capabilities of a design environment.

In short rather than a clutter of methods the various techniques can be neatly assigned their place in an ascending order of capability. The quality of a design environment is ranked in the same manner as the SBA Initiative, that is, the more comprehensive, integrated and interoperable its embedded tools then the better the environment.

The remainder of this section will describe these general principles of design environments and the remainder of the paper will then describe these methods culminating in a discussion of an implementation program.

3.2 Language Hierarchies

The simplest way to implement a design environment is to take a group of software tools related to a particular domain of design and merely embed them in a common command environment or so called ‘launch pad’. Such a design environment is only marginally better than the collection of tools.

To achieve high levels of interoperability a design environment (or development environment) must be implemented as an integrated whole. This begins with a disciplined, structured approach to the design languages and representations involved. A hierarchy of well defined (and behaved) separate languages is required [8].

Figure 5: Design Language Hierarchy (from [7])

Although separate the grammars and vocabularies of the language are related so that concepts in one language can be detailed or implemented in another. The whole system is a single architecture made by the integration of language translators.

4. PROGRAM & SYSTEM GENERATION

The generation of a complete design environment, which is structured as integrated languages, is best achieved by the use of a language translator generator [5].

Program generator technology whether for language translators or other software systems require as input a model or specification of the system to be generated [9]. The generator will then compose software components from a pre-existing library of components to produce the required application system.

Historically it was necessary to ‘hand craft’ the library of components. However, more recently well defined information models (such as formal grammars or the IDEF Process Models used to define Business Systems) are increasingly used to also generate the component libraries [10].

5. DOMAIN LANGUAGES AND MODELS

Business process modeling and software modeling both produce structured information models that can be used as the basis of system generation. Typically the definition of two interfaces or information models and mapping between them is sufficient to define, not only the input and output of a program, but also the processing it performs.

A well specified system will have information models for all of its formats such as documents, transactions, software, knowledge, etc. Information and process modeling such as this can be integrated as a language hierarchy by providing conceptual models. It is then possible to design, produce and maintain entire systems by automatically generating all software from these language definitions which are derived by modeling.
The term ‘language’ in this usage should not be thought of as speech, text or computer programming languages but refers to any system of parts that have well defined assembly options (vocabulary and its grammar being just one set of parts and assembly rules). Languages include graphical programs, concept graphs, CAD components and configurations of assemblies, etc.

Domain languages can represent all types of specialist knowledge and still be integrated into a language hierarchy. Domain experts can then provide knowledge to a system via a graphical domain language interface which is created from modular language descriptions, which can be specified as concept graphs by a systems language expert.

Each domain expert can communicate in their language of choice so long as that language is related to the systems language hierarchy. This brings us to the subject of semantic integration or how concepts in different languages are unified.

6. SEMANTIC INTEGRATION

Languages are not only defined in terms of their vocabulary and grammar (ie. components and structural rules) but also semantically, that is, by the meaning conveyed in sentences (ie. structural configurations).

In the operational semantics of programming languages the semantics is taken to be the function (behavior) of the program when it executes. The semantics of source code is defined in machine code and the semantics of the machine code is defined by the computer on which it executes. Equivalent semantics is equivalent behavior, so that source code programs written in different languages have the same semantics if their behavior is the same.

Defining the meaning of one language in the terms of another is a technique that can be applied to all forms language including those defining assemblies of parts. All that is required is that the definitions be precise for if they are not precise all sorts of inconsistencies and ambiguities will arise.

The task of defining meaning precisely is investigated in Ontology - a branch of linguistics, logic and philosophy. The linguistic approach, if only to consider meaning for translation between spoken and written conversation, is primarily concerned with agreement and terms can be defined in terms of a base or preferred language [11].

However, if specific situations and behavior are to be precisely defined then this base language must be logic or some other branch of mathematics. In modern logical ontologies semantics is mapped to (that is based on) a partially ordered mathematical structure called a lattice so that precise logical reasoning regarding the meaning, situations and behavior implied by language statements.

Figure 6: Top Categories in Logical Ontology

![Figure 6: Top Categories in Logical Ontology](reproduced from [11])

A semantically unified architecture defines both the syntax and semantics of all data and processes of which a language hierarchy is composed.

In practice information and process modeling is applied to define all formats for data, documents, spatial models, etc and these syntactic language and language instance are mapped one to another defining the semantics. Such definitions can be used to build a library from which software components and structures can be generated.

Figure 7: Logical, Domain & Application Ontologies

(reproduced from [11])

When programs and processes are defined semantically as a transformation (ie.mapping between two languages) and integrated into a language hierarchy then all processes in the system are hierarchically related.

7. SOFTWARE ENGINEERING

Current approaches to systems engineering are weak and do not result in architectures with hierarchically defined integration.

In order to define software for systems engineering

![Figure 7: Logical, Domain & Application Ontologies](reproduced from [11])

which is capable of logically composing or analyzing processes the vague and circular process definitions...
found in current systems engineering specifications must be ‘linearised’ (higher order methods that can cope with circular definitions are overkill for practical usage).

Methods now exist in both object oriented programming and logic programming which can incrementally locate and remove subtle circularities in data and processes that have been a dominant cause of systems malfunction. The software engineering processes of reflection, reification and refinement are now briefly described.

7.1 Refinement
When a statement of required functionality stated in terms which are more abstract is translated into an equivalent statement of function in more specific, concrete terms the translation or substitution is a Refinement. The abstract and concrete forms can be proven equivalent by the process of ‘bi-simulation’ [12].

The logical operation of refinement is essentially the same process of top-down design and hence can be used to model development and evolutionary processes.

7.2 Reflection & Reification
Reflection and Reification are operations between the various meta levels in a language hierarchy (or between a program or its meta program). Reflection occurs when a program interrogates its own meta data (ie allowed behavior) in its meta program (ie program having control of the first program). Reification occurs when the program requests its meta data and hence allowed behavior be changed and the meta program complies.

Refinement, Reflection and Reification techniques are combined in system synthesis software [13] that are able to re-structure their own processes and information structures in order to remove conflict and errors. Such systems synthesis techniques can be used to interactively build a system according to model definitions as they are entered by domain experts. Such systems can even detect and rationalize conflicting model definitions and consequently cope with domain experts evolving a language hierarchy on different levels concurrently.

Conflicting systems engineering specifications and defense development specifications can be ‘linearized’ and made rational ‘on-the-fly’ by using reflective system software which examines data object persistence across the system lifecycle [14]. These methods could be applied to the new STEP systems engineering lifecycle standards [15].

8. REVERSE ENGINEERING SOFTWARE
So far we have only considered generation of programs from a language hierarchy and improvement of a system by using a reflective architecture. It is also advantageous to reverse engineer software. Reverse engineering allows old software to have old interfaces upgraded and hence overcomes a limitation of current middleware methods.

Such software renovation, as the process is now called, can also fully document systems creating high level process models. The major limitation to software renovation is due to information loss when program generation occurs. This necessitates a human supervisor of program renovation. The process is similar to optical character recognition of text which has previously been printed at which time information was lost (see Fig 9).

7.1 Refinement
When a statement of required functionality stated in terms which are more abstract is translated into an equivalent statement of function in more specific, concrete terms the translation or substitution is a Refinement. The abstract and concrete forms can be proven equivalent by the process of ‘bi-simulation’ [12].

The logical operation of refinement is essentially the same process of top-down design and hence can be used to model development and evolutionary processes.

7.2 Reflection & Reification
Reflection and Reification are operations between the various meta levels in a language hierarchy (or between a program or its meta program). Reflection occurs when a program interrogates its own meta data (ie allowed behavior) in its meta program (ie program having control of the first program). Reification occurs when the program requests its meta data and hence allowed behavior be changed and the meta program complies.

Refinement, Reflection and Reification techniques are combined in system synthesis software [13] that are able to re-structure their own processes and information structures in order to remove conflict and errors. Such systems synthesis techniques can be used to interactively build a system according to model definitions as they are entered by domain experts. Such systems can even detect and rationalize conflicting model definitions and consequently cope with domain experts evolving a language hierarchy on different levels concurrently.

Conflicting systems engineering specifications and defense development specifications can be ‘linearized’ and made rational ‘on-the-fly’ by using reflective system software which examines data object persistence across the system lifecycle [14]. These methods could be applied to the new STEP systems engineering lifecycle standards [15].

8. REVERSE ENGINEERING SOFTWARE
So far we have only considered generation of programs from a language hierarchy and improvement of a system by using a reflective architecture. It is also advantageous to reverse engineer software. Reverse engineering allows old software to have old interfaces upgraded and hence overcomes a limitation of current middleware methods.

Such software renovation, as the process is now called, can also fully document systems creating high level process models. The major limitation to software renovation is due to information loss when program generation occurs. This necessitates a human supervisor of program renovation. The process is similar to optical character recognition of text which has previously been printed at which time information was lost (see Fig 9).
This analysis /translation /synthesis cycle provides means of migrating a current system to a new architecture while producing complete documentation. If combined with formal methods providing proof of correctness a high quality, virtually error free, system is produced.

9. STRATEGIC PROGRAMMING

Finally even more powerful methods are now available which generate programs from logical rather than process descriptions of their behavior. This allows AI theorem provers to reason about the program structure and produce highly optimal code that even the best programmers find difficult to match [17 – 22].

The techniques discussed so far can be integrated by adding stories to the language (meta level) hierarchy (see Fig 8) by providing logic extraction from processes via their representation as constraints. Process algebra can be used to analyze the system. Optimization is possible via efficient hierarchical constraint solving procedures.

10. IMPLEMENTATION

An incremental development program can proceed by constructing the pyramid of systems levels shown in Fig 6 one layer at a time from the bottom up. As these levels are constructed the data environment, process modeling, requirements & development simulation, simulation based design and then integrated synthesis environment are produced.

The implementation requires knowledge, generative and simulation teams.

REFERENCES

