Software Design for Reliability and Reuse

A Proof-of-Concept Demonstration

J. Bell, F. Bellegarde, J. Hook, R. B. Kieburtz, A. Kotov, J. Lewis,
L. McKinney, D. Oliva, T. Sheard, L. Tong, L. Walton, and T. Zhou*
Pacific Software Research Center
Oregon Graduate Institute of Science & Technology

The Pacific Software Research Center is developing a new method to support reuse and introduce reliability into software. The method is based on design capture in domain specific design languages and automatic program generation using a reusable suite of program transformation tools. The transformation tools, and a domain specific component generator incorporating them, are being implemented as part of a major project underway at the Oregon Graduate Institute of Science and Technology. The processes used in tool development and application of the method are being captured. Once completed, an experiment will be performed on the generator to assess its usability and flexibility.

This paper describes the SDRR method and illustrates its application to the Message Translation and Validation domain, a problem identified by our sponsors because of a previously existing state-of-the-art solution based on code templates produced by the SEI[10].

1 The SDRR design concept

SDRR is a method for the design of flexible and powerful software component generators. With a component generator, software components themselves are not the basis for design reuse—the design encapsulated in the generator is the reusable artifact. When a subsequent application or version of a design is needed, design modifications are made to the specification that is input to the generator, and a new software component is generated automatically. This allows each design change to be made at the appropriate level of abstraction—details of the software irrelevant to the change are not involved. It also allows the applications designer and the software maintainer to use a design language that directly expresses application domain concepts, rather than the encodings of these concepts necessary in wide-spectrum programming languages.

This generation based method can be viewed from two fundamental perspectives. The developers of a generator see a reusable set of tools and a method for their use in the construction of generators. The users of a generator see a system that allows them to go from a specification to a software component without having to understand the details of how the generator is constructed.

Component generators achieve their greatest advantage for the design of families of software modules that are needed in many particular instances. If a software component is anticipated to

*The authors were supported in part by a contract with Air Force Materiel Command (F19628-93-C-0069).

be a one-off instance, dissimilar to any existing design, then developing a component generator to produce it has little advantage. However, this is almost never the case. Most software modules have family resemblances to other modules and undergo use and modification as part of an extended life cycle that requires their design to be maintained and updated. SDRR generators are intended to produce software components that can be reliably and inexpensively maintained throughout their entire life cycle.

An SDRR generator design team includes two kinds of specialists who need to coordinate their efforts but are able to work concurrently and independently on their assigned tasks.

- The domain experts who have overall responsibility for the software design, its validation and documentation.
- The SDRR method experts, who have knowledge of the principles of language design and are familiar with the transformation tools. (In the method definition report, these roles are significantly refined[6].)

Steps in the design of a software component generator

The design of an SDRR component generator proceeds in a series of steps. The first step needs the attention of the domain experts. In this step, domain analysis is performed to determine the requirements of the intended application. This step is common to all software development. It is not unique to SDRR. The other steps are unique to the SDRR method. Method experts must:

- Formulate a domain-specific design language (DSDL) in which to express the parameters, operations and constraints necessary to meet the requirements of the domain application.
- Formalize a computational semantics of the DSDL in terms of ADL, the algebraic design language used in SDRR[8, 7]. This task requires a computer scientist with advanced training in formal aspects of programming languages and software design.
- Design an implementation template characterizing the execution environment. SDRR implementations are stereotyped. A developer designs a set of implementation primitives that specify how the computational semantics of the DSDL are to be realized in terms of the target programming language Ada.

Performance improvement is obtained through the use of automated program transformations that are applied during the course of program generation. These transformations are mathematically based and are guaranteed to preserve the computational meaning of the ADL-specified semantics. The transformation tools include: HOT, which applies higher-order transformations[12, 13]; Schism, a partial evaluator[5]; Firstify, an implementation of Reynold's algorithm for defunctionalization[11, 1, 2]; and, Astre, a first-order transformation tool based on term-rewriting techniques[3, 4].

When an SDRR-designed program generator is applied to a DSDL specification, it automatically applies the necessary transformations. The transformation tools provide a very

advanced optimizing compiler that takes ADL as input and generates Ada as output. The system architecture is summarized in Figure 1.

2 Domain Specific Design Languages — DSDLs

A domain specific design language is intended to formally specify software designs. It is a formal language that is expressive over the abstractions of an application domain. A DSDL may be wholly or partially declarative or it may be a functional language with libraries of functions specialized to the application domain. Common examples of DSDLs are:

- Schema description and query languages for databases.
- Layout languages for prettyprinting the text of computer programs.
- A message format description language for the message domain of military C^3 systems (MTV).

Using a DSDL, a domain expert can express concepts in the problem space directly, without encoding them. This allows the domain expert to formalize the specification of a software solution immediately instead of communicating a specification informally to a software specialist who may be less familiar with the intended application.

The definition of a DSDL introduces a new, formal, reviewable artifact into the domain analysis phase of a software development process. This formal definition may benefit the process just as developing formal specifications benefits the requirements analysis phase.

2.1 Designing a DSDL

A DSDL is defined by a computer scientist in consultation with a domain expert. In the design of a DSDL, a dialogue is necessary between the two in order to settle the following three important issues. The issues are illustrated with specific examples from the DSDL for the message translation and validation domain (MTV), which will be described in greater detail in Section 6.

- To clearly identify the principal conceptual abstractions of the domain.

  For the MTV domain the essential abstractions are the internal and external representations and the translation functions that map between them.

- To formally define a language of terms to represent these abstract concepts. A term language can be defined in terms of a syntactic phylum (syntactic category) for each conceptual entity. These languages give a means to express instances of the concepts and the relations between them.

  For MTV, the basic phyla describe the logical structure of the internal representation of data and the message translation actions that parse (and implicitly generate) a message.

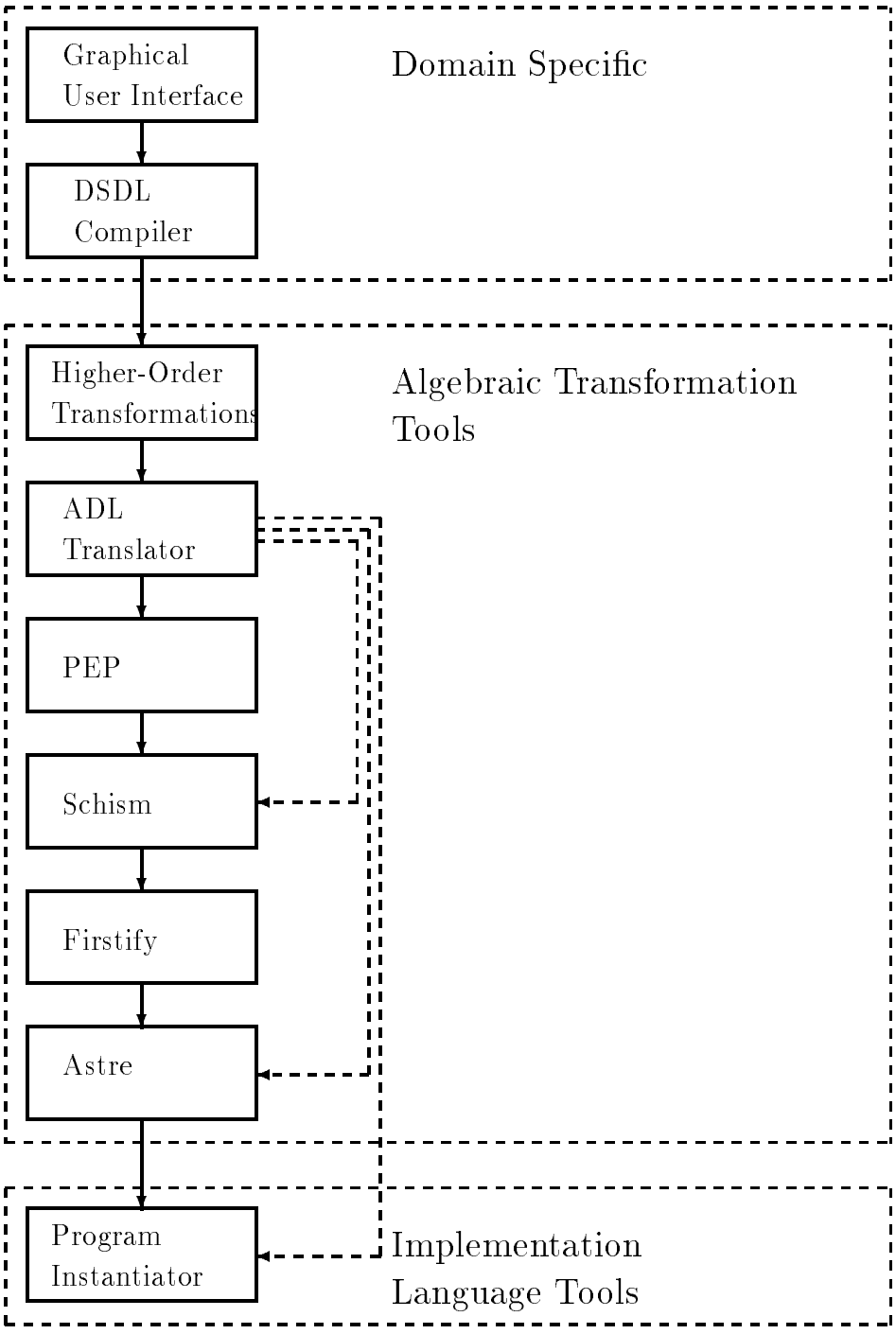

Figure 1: Software Architecture of an SDRR generator

- To interpret the relations among the principal conceptual entities. This interpretation is initially given by the domain expert in an informal manner, by describing the relations in natural language (English). The computational content of this description will later be elaborated by giving a formal semantics to the DSDL.

  In MTV, two documents generated at this stage of design have been delivered to the clients. The first was a specification document summarizing our domain analysis, including the identification of the principal abstractions[9]. The second was the preliminary definition and informal semantics of the language[16]. Documents giving the revised DSDL definition and its formal definition in ADL are planned.

Often, a graphical user interface (GUI) can be used to help an application designer formulate a design in the DSDL. With a well-designed GUI, the application designer does not need to "learn another language" to use the DSDL. Even when a GUI is not developed, DSDLs tend to be quite small and readily learned since they include only those concepts necessary for the problem domain.

The need for a GUI is determined by studying the natural flow of work for engineers using the generator. For MTV, this determination is not yet complete; preliminary indications are that a text-based interface will fit most naturally into the work environment. A GUI, however, may allow the environment to evolve to include less skilled workers.

3 Formalizing the semantics of a DSDL

The formal semantics of a DSDL is defined in terms of PacSoft's algebraic design language (ADL)[7, 8]. This semantics gives the DSDL a computational interpretation in which the relations between the principal concepts of a design abstraction are formalized.

The first step in the formal specification of a semantics is to specify a datatype that corresponds to the abstract syntax of the DSDL. To each operator of the abstract syntax there will correspond a data constructor of the datatype. The semantics of a term constructed with a given data constructor will be composed from the semantics of the subterms given as arguments to the data constructor.

The control structure of an ADL program is specified through families of high level combinators. For the MTV example, the translations are specified in terms of about a dozen primitives and five basic combinators. From these basic building blocks, arbitrarily complex translation functions can be constructed.

Each operator of the abstract syntax of a DSDL is given a computational interpretation by a semantic function. The semantic function is well-typed in the type system of ADL, and is defined by cases on the data constructors of the ADL datatype derived from the abstract syntax of the DSDL. For each such case, the prescribed meaning of a DSDL fragment is specified by a computation programmed in ADL. In the case of MTV, the semantic functions are defined by approximately three pages of ADL code.

This programming technique uses the syntax of the DSDL to structure the specification of a computational solution. The resulting solution is compositional; less attention is given to the

efficiency of a solution than to the regularity of its construction from its component parts. The goal is to specify a computation in such a way that it is amenable to formal reasoning, so that one can verify that it corresponds to the informally specified problem requirements. Algorithmic efficiency will be improved at a later stage by meaning-preserving program transformation of an ADL specification and by compilation into an efficient representation in Ada.

4 Transformational Improvement

When the semantics of a DSDL is fully elaborated in ADL it is algorithmically effective. A component design specified in the DSDL can be executed as a rapidly constructed prototype. However, without further work, it is likely to have poor performance in terms of execution time and space usage. The SDRR method encourages highly modular design of semantic functions in ADL. This produces a design that is easy to understand, to validate and to maintain, but engenders many more uses of function composition than might otherwise be necessary. Accordingly, control structures that might be shared are often duplicated, and intermediate data structures may be built and analyzed when they could have been avoided by careful programming.

To avoid paying performance penalties for modular design, SDRR employs extensive program transformation on the ADL specification. The transformations that are used are *meaning preserving*, which implies that they will never introduce errors that were not present in the original design. These transformations are, in fact, derived as instances of theorems in the algebra of ADL. There are transformations that support:

- deforestation—elimination of intermediate data structures;
- fusion—consolidation of similar control structures;
- accumulator introduction—caching of values to avoid recomputation;
- recursion elimination, in favor of iterative control;
- introduction of state (i.e. global variables).

Transformations of an SDRR design are applied automatically. Transformations are directed by pattern-matching which triggers the invocation of embedded tactics.

5 Implementation Templates

An implementation is specified by a set of *implementation templates* and an interface specification[14, 15]. An interface specification documents the (typed) system interface that will be seen by the software component that is the object of the design. The functionality required of the interface can be specified informally or in terms of a first-order logic or software specification language.

The interface provided by the designed component includes the types of its visible functions or procedures, together with the formal specification of the component as elaborated in the design.

Implementation templates are macro-like translation forms for the primitive access and construction functions of ADL datatypes. For example, the template mechanism is used in the MTV domain to make operations on bit-strings visible in ADL. A set of implementation templates must contain generic templates for algebraic datatypes but it may also contain specialized templates for specific types that are commonly used. Through implementation templates, a designer can specify a hashed symbol table, for instance, as the implementation of a dictionary.

Implementation templates are typically quite small, of the order of a few hundred source lines, although a set of templates can grow if additional, specialized implementations are specified for particular datatypes. These templates are highly reusable, both because templates are copied many times during the translation of a single design from ADL to the target implementation language, and because a set of templates can be used in any number of specific applications.

6 MTV: An Example

A message translation and validation problem is presented to an engineer as an *interface control document* (ICD). The ICD is a semi-formal specification of the external representation of the message. It consists of general information, such as message size, followed by a field-by-field description of the message and its contents. Field descriptions may themselves have internal structure. For example, a date field will contain a day, month and year. Some fields may have different kinds of data, for example a field may represent an altitude if it contains only digits or a location if it contains alphabetic characters. The ICD also contains constraints on valid messages; these are expressed in informal (and sometimes ambiguous) English language annotations.

For each message format, the engineer must design two additional representations: (1) an internal representation to be communicated to other system components and (2) a "user" representation to be used in system logs and test message generation. A "solution" consists of six functions: translation functions between external and internal, between user and internal, and check functions on user and external messages.

In the specification document, Lewis introduced a "logical" representation that is essentially the internal representation without the intrafield constraints[9]. This provides a representation that can be tested for compliance to the constraints. In the MTV DSDL a user specifies the logical structure of the message as the logical type. The user also specifies one direction of the translations from logical to user and from external to logical. From these specifications, the system infers the inverse translations as well. Below we will focus exclusively on the external to logical translation; we will not consider the logical to user translation.

A typical problem specification is given in Figure 2. Study of the domain analysis in the SEI model solution[10] and analysis of messages led Lewis to specify that the basic types in messages are integers, integer subranges, strings, string subranges, booleans and enumeration

No.	Field Name	Size	Range	Amplifying Data
1	Course	3	001-360	In degrees.
			000	No value reported.
	Field Separator	1	/	Slash.
2	Speed	4	0000-5110	In knots.
	Field Separator	1	/	Slash.
3	Altitude or	0 or 2	01-99	In thousands of feet.
	Track Confidence		HH	High confidence.
			MM	Medium confidence.
			LL	Low confidence.
			NN	No confidence.
			Blank	No altitude value reported or altitude less than 1000 feet.
	Field Separator	1	/	Slash.
4	Time			Time Group
		2	00-23	Hour
		2	00-59	Minute
	End of line	1	CR	Carriage return.

Figure 2: Sample Interface Control Document

types. These types may be arranged in records (labeled products), variant records (labeled sums), lists of arbitrary length, or arrays. This analysis determined the logical type structure of the DSDL for MTV.

The message reader actions are constructed compositionally using the structure present in the type system. A set of primitive translation functions are provided for the base types. Rules for combining products, sums, arrays and lists of actions are used to aggregate a collection of actions into a new action. For example, `Asc2Int 2` reads two ascii characters (which must be digits) and produces an integer value. Two such read actions can be aggregated into a record reader by writing them inside curly braces:

```
{ latitude : Asc2Int 2, longitude : Asc2Int 2}
```

To accommodate variant records, and to make the reading of enumeration typed expressions more consistent, a failure mechanism is provided. Consider the Altitude or Track Confidence field in the example. If the data is presented as a sequence of digits it is assumed to be an altitude, but if it is alphabetic it is a track confidence. If it is neither of these, it must be empty. In all cases the message is delimited by a "`/`". The failure mechanism supports this by allowing variant record readers to be declared within square brackets as follows:

```
[ Altitude : Asc2Int 2,
  Track_Confidence: to_Confidence,  (* a previously specified action *)
  No_value_or_Alt_less_than_1000: Skip 0] @ delim "/"
```

In this case, first `Asc2Int` tries to read the input. If it succeeds, the variant record reader constructs an Altitude. If it fails, the `to_Confidence` reader is invoked. Again, success of the component reader will lead to success of that branch, and failure will lead to attempting the next branch. If all branches fail the variant record reader fails. If the variant record reader succeeds, the delimiter reader is invoked. The delimiter reader succeeds only if it reads exactly the string specified as the delimiter. Thus, the aggregate reader succeeds exactly when the input is an appropriately delimited field as specified in the ICD.

In a similar manner, the other fields in the ICD may be translated into the formal notation of the MTV DSDL program in Figure 3. Note that this translation is very straightforward—there are no unnecessary encodings of concepts. The MTV DSDL "program" looks more like a specification than an algorithm. Maintenance of an artifact expressed at this level is expected to be significantly easier than maintenance of a code level representation.

The MTV DSDL translator takes a specification at this level and compiles it into an intermediate form in ADL, the algebraic design language. A series of program transformation tools optimize this, removing any layers of interpretation introduced by the generality of the solution. Finally, the program instantiator generates Ada code. This code can then be integrated into a command and control system.

7 The Project

The SDRR proof of concept demonstration project is unique in two important ways: first, the goal of the current effort is the support of a carefully planned software engineering experiment comparing the new technology to an existing technology; second, aggressive project management methods are being applied to an academic team doing research.

In the planning phase of the project, PacSoft negotiated an experiment with the sponsor to test the claims of usability, flexibility, predictability, productivity and adaptability. In this experiment, two subjects provided by an independent contractor will do a matched series of tasks selected by the sponsor in each of the two technologies being compared. Data on their performance on these tasks will be collected and analyzed. The results of this experiment give both a success criteria for the current effort and will help identify directions for future research.

Recognizing that the "chaotic" methods traditionally applied in one and two investigator academic research projects would not scale to a team of over a dozen researchers, PacSoft made a conscious decision to use more mature management practices. We began with the development of a plan for the research project, developed jointly with Paul Szulewski, Faye Budlong, Walter Ellis and Stuart Schiffman at Draper Laboratories. This has been followed by a systematic maturation of the group as we build ownership, define, document and evolve our processes. To provide feedback to these maturing processes, and to give the clients evidence of the team's progress, we have integrated the collection and analysis of metrics data into our management, planning and assessment. The keen interest of our clients and the quarterly

```
(* Type declarations *)
type Confidence_type = [High, Medium, Low, No];

type Alt_or_TC_type = [Altitude: integer(1..99),
                       Track_confidence: Confidence_type,
                       No_value_or_Alt_less_than_1000];

type Time_type = {Hour: integer(0..23),
                  Minute: integer(0..59)};

message_type MType = {Course: integer(0..360),
                      Speed: integer(0..5110),
                      Alt_or_TC: Alt_or_TC_type,
                      Time: Time_type};

(* Action declarations *)
EXRaction to_Confidence = [High: Asc 2 | "HH",
                           Medium: Asc 2 | "MM",
                           Low: Asc 2 | "LL",
                           No: Asc 2 | "NN"];

EXRaction to_Alt_or_TC = [Altitude: Asc2Int 2,
                          Track_confidence: to_Confidence,
                          No_value_or_Alt_less_than_1000: Skip 0
                         ] @ Delim "/"; (*  field separator "/" *)

EXRaction to_Time = {Hour: Asc2Int 2,
                     Minute: Asc2Int 2
                    } @ Delim "\n";     (* CR as field separator *)

EXRmessage_action to_MType = {Course: Asc2Int 3 @ Delim "/",
                              Speed: Asc2Int 4 @ Delim "/",
                              Alt_or_TC: to_Alt_or_TC,
                              Time: to_Time};
```

Figure 3: Sample MTV DSDL specification

review process that we have put in place help keep the group invigorated and focused on our common goals. To date, our experience with these management practices has been positive.

8 Conclusion

The next generation of software tools will support the manipulation of designs directly without requiring the manipulation of intermediate encodings of these concepts in programs. The SDRR proof-of-concept demonstration project is providing:

- A method for developing the appropriate domain specific design languages and implementing flexible and maintainable generators supporting them;
- A tool suite to support this method;
- A generator for a real-world problem constructed by applying the SDRR method;
- An experiment comparing the resulting generator to the current state-of-the-art; and,
- A record of the process and metrics data characterizing our experience with the method and its development.

Together, this combination of research, demonstration, and experimentation exemplify a new paradigm for the rapid transfer of technology from an academic research institution into industrial and government software development practice.

References

[1] Jeffrey M. Bell. An implementation of Reynold's defunctionalization method for a modern functional language. Master's thesis, Oregon Graduate Institute, January 1994.

[2] Jeffrey M. Bell and James Hook. Defunctionalization of typed programs. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, February 1994.

[3] Françoise Bellegarde. ASTRE, a transformation system using completion. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, 1991.

[4] Françoise Bellegarde and James Hook. Monads, indexes, and transformations. In *TAPSOFT '93: Theory and Practice of Software Development*, volume 668 of *LNCS*, pages 314–327. Springer-Verlag, 1993. A page was omitted from the proceedings, it may be obtained via ftp from `ftp.cse.ogi.edu` in the file `pub/pacsoft/papers/tapsoft.dvi`.

[5] Charles Consel. The Schism Manual, version 2.0. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, 1992.

[6] Richard B. Kieburtz. Software design for reliability and reuse (preliminary method definition). Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, October 1993.

[7] Richard B. Kieburtz and Jeffrey Lewis. Programming with algebras. Technical Report (submitted for publication), Oregon Graduate Institute, October 1993.

[8] Richard B. Kieburtz and Jeffrey Lewis. Algebraic design language (preliminary definition). Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, January 1994.

[9] Jeffrey R. Lewis. A specification for an MTV generator. Technical Report 94-003, Department of Computer Science and Engineering, Oregon Graduate Institute, September 1993.

[10] Charles Plinta, Kenneth Lee, and Michael Rissman. A model solution for C^3I message translation and validation. Technical Report CMU/SEI-89-TR-12 ESD-89-TR-20, Software Engineering Institute, Carnegie Mellon University, December 1989.

[11] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM National Conference*, pages 717–740. ACM, 1972.

[12] Tim Sheard. Type parametric programming. Technical Report 93-018, Department of Computer Science and Engineering, Oregon Graduate Institute, November 1993.

[13] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Proceedings of the conference on Functional Programming and Computer Architecture*, Copenhagen, June 1993.

[14] Dennis Volpano and Richard B. Kieburtz. Software templates. In *Proceedings Eighth International Conference on Software Engineering*, pages 55–60. IEEE Computer Society, August 1985.

[15] Dennis Volpano and Richard B. Kieburtz. The templates approach to software reuse. In Ted J. Biggersstaff and Alan J. Perlis, editors, *Software Reusability*, pages 247–255. ACM Press, 1989.

[16] Lisa Walton and James Hook. A preliminary definition of a domain specific design language for message translation and valiation. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, February 1994.