A Proposal for a JSP Executable Specification

Alan Richardson

October 21, 1994

1 Executable Specifications

1.1 definition

A specification is 'an abstract, computer-oriented representation of a set of user requirements' [RATCLIFF 87]. Obviously there are many levels of abstraction in a typical design, and these are reflected by the many different specifications in use throughout the design of the system.

TAXONOMY

Sommerville [SOMMERVILLE 85] states that 'software specification and design are so closely connected that it is impossible to separate them.' With the onset of automatic code generation tools and executable specifications this can be taken further; until we witness the mergence of specification, design and implementation into one extended mulit-step process which feeds forwards and back on itself while the specification is modified to produce a number of different levels of specification which can be validated back to the original specification level.

Intellectually this process can be split into individual sections such as design and implementation , and further into functional specification, performance specification and modular specification, but this approach is dangerous and leads to independent specifications which require conceptual leaps to move from one to another.

1.2 Problems with specifications

Traditional program development is carried out by first doing an analysis of the problem, creating a problem specification and giving this to a programmer to write. Each of these stages will have its own design method, or possibly not, and these may not be abstractions of one another but interpretations of each other. Liskov and Berzins describe a program specification as being "a precise and independent description of the desired program behaviour" [LISKOV & BERZINS 79]. If a program specification is indepedent then it was

presumeably derived by interpreting a preceding analysis and design specification, and this can lead to the traditional problems associated with program development - primarily, developing the wrong system, and developing it badly - which arise from incorrect specifications.

To make matters worse, the traditional approach is often to create less precise program specifications in order to avoid upsetting the programmers. ".. every programmer I've met would bristle and seethe if you told him how to do his job. And quite rightly. Desiding the most appropriate way to program what is required is the programmer's job; therein lies the creativity of programming" [MEILIR 88]. When a program specification is precise enough then there is very little creative effort involved in writing the code, it can in fact be automated. The creativity of the development task lies in producing a detailed and correct specification which can be abstracted back to its origins. Creative steps, fueled by imprecise objectives are to be avoided if the analysis and design effort is not to be wasted.

The ideal specification solution is one where the same specification techniques for specifying the requirements are used throughout the specification process right up to code production. This specification panacea unfortunately does not yet exist, and this leads to a major split in the specification process; requirements specification and system specification, where one, or possibly more than one, technique is used for requirements specification and another for system specification. Informal specification techniques such as Jackson Structured Design (JSD), which can specify to code level, can be used for requirement specification in the form of Structured diagrams, but it is a matter for debate whether these can be understood by the user. Formal specification techniques, which again can be taken to code level, are unlikely to be understood by the user because of their mathematical basis.

Specifications should be communicable, not to everybody but to the person whom they are aimed at; a user should not be expected to understand the same specification as a systems analyst but both specifications should detail exactly the same thing, but at different levels of detail.

This quest for clarity has led to guidelines which tell us to avoid optimisation, this is perfectly reasonable since it may require the alteration of specifications from the requirements. If this can be done automatically then this would be an ideal solution and the research into program transformation systems is important [PARTSCH 90], these are principally based around mathematical specifications to enable them to be easily transformed into efficient systems. Unfortunately these have not yet reached the state of full automation.

Informal specification strategies such as Jackson Structured Programming (JSP), and Structured Systems Analysis Design Methodology (SSADM) should avoid optimisation in order to remain true to the initial specification. This becomes important when modifications have to be made to the system and maintenance is carried out.

The traditional process of using specifications as guidelines for programmers,

results in a large amount of maintenace problems. A change in a specification is effected by altering the source code, this can lead to inconsistencies in, and between, the specification and the source, and can lead to errors being added to the program. This can be avoided if the source can be directly generated from the specification.

There are many different classifications within which specifications can currently be placed, the two most important distinctions as related to this document are; executable and non-executable, formal and informal.

Executable specifications are those that can be directly executed, these contain enough information that they can be computable. Non-executable specifications are those that are typically used for requirements or conceptual analysis such as Entity Relationship (ER) diagrams, Data Flow Diagrams (DFDs) and Z.

Formal and informal specifications refer, respectively, to those which are based around mathematics, and those which are not. Informal specification techniques instead tend to be based around diagramatic notation.

1.3 Informal Specification Techniques

There are a large number of informal specification techniques available and the following examples are not intended to be a complete summary. They have been chosen in order to illustrate the range of methods available and highlight some of the problems associated with them.

Informal techniques fall into two categories, those that are used to help the analyst come to terms with the semantics of the situation, these include, ER diags, Object Orientation (OO) and DFDs. Others are more rigorous and offer a basis for code generation because they contain more computable information such as operations or time structures, these include Entity Life Histories (ELHs), and structure charts. Others specify the algorithmic complexity of the situation in sufficient detail that full program code can be mechanically created; these include JSP, psuedo-code, and flow charts.

Natural Language Natural language is the most obvious means of specifying any problem, and for this reason is the usual first step in the specification of a project. The obvious advantages of this specification technique is that it can be understood by anyone involved in the project. The main problems are that it is ambiguous, verbose and unlikely to be complete[DROMEY 89].

Pseudo-code & Structured English - Pseudo code and structured english are used for program specifications. Normally used when the programmer is interpreting a vague specification and as such are open to misinterpretation.

SSADM An SSADM specification encompasses the design and analysis aspects of a systems life cycle, and uses a number of smaller specification techniques within it - ER diagrams, DFDs, ELHs. This involves a lot of independant specifications that can be difficult to translate into programs, the intention being to provide programmers with free reign on the interpretation. This results in less detailed program specifications being produced.

Object Orientation OOD does present some problems during the implementation phase because it does not provide 'mechanisms for defining algorithms and representations of data structures' [ORMSBY 91], and neither does it provide a mechanism for structuring the actions into a time sequence suitable for a program.

Jackson Structured Programming JSP is a data driven graphical specification language. The main steps in the development of a JSP specification are:

1. Model data structures
2. Form program structure from the data structures
3. List and allocate the operations and conditions
4. Translate into program code

From the above steps it can be seen that JSP is a series of refinement steps of data into programs. This differs from other informal techniques such as SSADM which use a large range of conceptual modelling techniques which cross reference to provide a program specification.

The JSP specification contains the source code text as operations and conditions and as such contains enough information to automatically produce full programs. In this respect JSP is a rigorous specification technique, the actual specification produced becomes the basis for the source code, there need be no intervening interpretation steps. As a result every JSP specification is a clear and unambigous solution to a given problem.

JSP uses simple diagrammatical conventions with very few constructs, the basic constructs are simple to learn and can be understood very quickly. This allows the technique to be quickly and easily taught to programmers of very different abilities.

It used to be true that one of the disadvantages of JSP was that in order to produce complicated programs, it was necessary to use more complex techniques; for example inversion and correspondence. These were seen as difficult to code, and the technique of correspondence would take precedence during the design of a program because although it was the least rigorous method of producing code, it was the simplest to code. However

with the aid of an automated package the process of inversion becomes simple, and the more preffered method since it allows greater scope for maintenance, and module re-use.

The documentation produced by the JSP specification technique is sparce when compared to a technique such as SSADM. However this may not be a disadvantage since the specification in JSP is constantly up to date (assuming code generation is being carried out) and this may not be true of an SSADM designed project where the large number of initial specifications (ER diagrams & DFD's) may not be updated when and if the program changes. Also to quote [LISKOV & GUTTEY] "Generally as a specification grows longer it is more likely to contain error, less likely to be completely and carefully read, and more likely to be misunderstood."

The main feature in common with the informal specification techniques is that they tend to use a highly graphical notation (ER, DFD, JSP, SSADM, STTs). This allows them to be used as a communication tool between the user and the analyst. As a communications medium they are not perfect and this has led the to the use of prototyped solutions where the user can see the system in question. However when no prototyped solution exists they are more attractive to the user than the corresponding mathematical notation of formal specification languages.

1.4 Formal Specifications

Formal specifications have a formally specified syntax and semantics, a number of them are directly executable e.g OBJ, others need very little modification to allow them to be implemented in a functional language e.g VDM, and some like Z are a mechanism to think about the problem and to prove the consistency of the solution without concerning the analyst with implementation details although they can be prototyped in languages such as prolog.

Because they employ mathematics the specifications produced can be proved consistent and syntactically correct. However proof of correctness does not mean that it is correct from the users perspective, the specification still needs to be tested in order to validate that it matches the users requirements and that all special cases have been identified.

The emphasis on mathematical techniques has led to a high learning time being associated with formal methods, and has contributed to a small user base in the commercial DP environment where the staff have little or no mathematical training.

A lot of analyst work must be spent on proof obligations in order to carry out formal specifications correctly.

Very few tools are available to help the analyst with formal methods other than documentary ones, although syntax checkers and proof support tools are becoming available, for example the fuzz type-checker for Z.

These specifications are often used for the specification & production of rapid prototypes, they are not intended for the production of a final version, "these languages are oriented toward verification and cannot yet support completely automated translation to an efficent executable form" [KAISER et al]. To produce a final version of the system in a conventional programming language the specification has to be transformed into an efficient version or coded - easily but inefficiently - using recursion.

1.4.1 Formal vs Informal Notation

Formal specification methods use the concise and unambiguous notation of mathematics. This can make them communicable without error to other practitioners who understand the notation, but the notation itself can be difficult to learn and the design is uncommunicable without the relevant experience. This is also true about the rigorous informal specification techniques such as JSP, however the informal techniques have a shorter learning curve and can be taught without the background knowledge necessitated by formal specification.

Many informal techniques are based upon graphical modelling methods and these, dependant upon the methodology, can be ambigous in many ways e.g Structured Analysis techniques such as DFD's and ERD's which are open to interpretation.

Informal diagramatical techniques can be difficult to code from although unambigous informal design notations do exist, notably JSP, and these detail exactly what the program will do. The only interpretation involved with JSP is moving from the specification to the code when no automation is available.

Informal techniques do lend themselves to communication with the less technical staff on a project such as the managers and the users, and anything that is not understood immediately can be explained very quickly.This allows the specification to be validated that it meets the users requirements more readily than a formal specification.

Although such graphical specifications can be used in order to aid the communication of ideas between the analyst and the user they may not be adequate in determining that the user requirements have been met. The reasons behind this are the reasons that have pushed forward protoyping techniques as a specification and design tool.

2 Prototyping

The problem with both formal and informal specification techniques is that although thay help to build more correct systems, these systems may not be doing the correct task. This leads to an increase in maintenance effort related to the project, in order to produce the system that should have been produced in the first place.

These problems occur for two reasons [VONK 90]; the diagrams produced are not as accessible to the users as the analyst believes, and because the users needs do not remain stable after the requirements definition process.

The communication problem is important in relation to informal specification techniques because these techniques are intended to bridge the understanding gap between the user and the analyst. This is not so with formal specification methods because the user is never intended to be able to understand the completed specification - this why it is difficult to verify a formal requirements specification.

The process of requirements specification causes users to think about the problem carefully. From a user point of view this process is difficult [SPENCE 91] because they don't know what a computer can offer them, and they do not know how it will affect their working practices. Consequently as development proceeds, the user begins to see the potential of the computer and changes the system requirements.

With typical specification methods the user is required to sign off the specification. This allows the analyst to refine this specification to such a degree that the program can be produced. If the users requirements alter then it can be dificult to change the specification to match this. Prototypes are intended to facilitate this kind of change easily.

Prototyping can validate that the analyst understands the customers needs, but it is useful useful only if it can be done quickly. Prototyping is not a panacea for requirements specification, it does have its problems,and these notably involve the use of the prototype, or a slightly modified version of the prototype, as the final system, this is not likely to be efficient, or well error trapped.

3 Specifications in General

The importance of specifications is well known. They serve as a complete and detailed definition of the user's requirements, without which any program developed will fail to become the desired product. Altering the program to become the ideal will involve costly and late changes thereby resulting in a software product of possibly poorer quality.

Doing any job correctly requires the use of the correct tools, failure to do so will result in a job with loosely fitting components ready to fall apart or joints that are too closely knit to allow sufficient movement, the same is true with software development. Getting the specification correct requires the use of the correct method; the method chosen should be dependent upon the task to be specified, the staff involved in the specification and the resources available to support it. For example certain techniques have proven their use in a DP environment e.g JSP and SSADM, these however are unlikely to be suitable for the development of a life critical application such as automatic pilots and the use of formal methods to attempt to prove the correctness of the software may

be advisable. The experience of the staff should be taken into account when choosing the specification method, if staff are trained in a method that is suitable then it should be used. If a new method is to be used then the staff should be adequately trained, and given time to develop experience in the specification technique. If the staff is of limited experience with the specification technique then the technique should be stuck to with 100% dedication, deviating from the technique when the practitioner has had little experience with it may generate problems that otherwise would not have occured. Similarly; mixing different specification techniques in order to communicate the requirements effectively should only be done when the practitioner is familiar with at least one of the techniques.

What makes a good specification is a matter for debate but Balzer and Goldman [BALZER et al] propose 8 principles; intended to be applied to formal specification techniques, but applying them to specifications in general results in the following picture of a specification. It should describe what is done, but obviously this should be extendible to eventually describe how something is done in order to be operational. It should be used to analyse the environment, and take account of how the system reacts with the environment and how the systems individual parts interact. It should be used as a means representing the user view by modelling real world objects, and it must be easily changed to reflect the dynamic nature of systems and design.

James Martin [MARTIN 85] provides his own list of desirable properties which a specification should have, and these emphasise; simplicity, clarity, abstraction, automation and execution.

Obviously both sets of principles are geared towards different specification language groups; Martin's are aimed at 4GLs and Balzer's at formal languages. However both stress that a good, or as Martin would say a 'Proper', specification should be viewed from different levels of complexity, allow change, be understandable and be computable.

Both take a different view of just how much computable information should be present in a specification, Martin obviously believes that full computability should be possible but that it should be hardware independent. Balzer et al take the stance that it should be computable only in the most minimal instance, for example via animation, and possibly only theoretically.

These two views on computation stress mainly from the way that the techniques are carried out, Martin uses 4GLs to produce a specification that dictates automatically what the completed program should do. Balzer et al use a specification which is to be used as a guide for the completion of the finished program and as a means of validating the finished code against the specification.

A Specification need not be directly executable, or taken to a level of complexity where it is possible to generate code - they may be used simply to think about and gain a clearer understanding of a situation for requirement specification purposes. This allows for experimentation, by trying out various solutions without having to implement them. One of the desireable properties

of a specification as suggested by Balzer and Goodman is that a specification should outline *what* should occur rather than *how* it should be done. With non-executable specifications this causes problems when moving from the specification to the code. This can often be a difficult process and there is a very real danger of errors being introduced during the conversion period.

Verification of the programs produced using non-executable specificatons involves comparing the code with the specification; something that can not be done when the specification is either executable or can be directly converted into code. To quote [LISKOV & BERZINS 79] "... a program should not be its own specification, because this eliminates the redundancy needed to make verification meaningful". In these cases, where the specification itself is the program, there must be a way of checking that the specification is correct before it is released to the users. If the specification was developed by refinement then backward validation should be possible.

Executable specifications allow user validation of the specification before code production begins in earnest and the system is still flexible and ameanable to change.

Any specification that can be automatically turned into code implies that the same structure is shared by both specification and implementation; leading to an easily maintainable system where all changes made to the specification are reflected accurately and easily in the implementation. This also implies that the specification itself should be directly executable although this is rarely the case.

4 Execution of JSP

Out of all the possible methods of specification, why pick JSP as the candidate for execution?

- It is widely used in the DP industry for the development and maintenance of COBOL programs.
- It can be automated to produce complete COBOL programs
- It can be communicated between designers and users
- It produces accurate documentation
- It can be easily taught

There exist graphical tools such as PROBENCH, JSP-tool, Speedbuilder, and PDF which automate the production of diagrams and code. This allows the analyst to become the programmer and the finished product does not have to be specified in a less complete form to avoid offending any other craftsmen.

Any JSP development is concerned with the creation and updating of detailed program specifications. Any errors therefore are specification errors however in order to test the validity of a JSP specification the current practise is to generate and test code. This is aided by the numerous JSP packages available which deal with the translation of diagram into code that would otherwise be done by the human programmer.

FIGURE 1

Figure 1 represents the typical process undergone when producing a JSP program structure diagram (PSD). The PSD is created and turned into code. This generated code must be compiled and then executed in an environment other than the diagrammer. This entails the use of the compiler's source level debuggers, this is not a straight forward execution of the diagram but is the execution of the source code generated by the diagrammer.

The problem here is that the emphasis moves away from JSP being a specification and design technique, concentrating on the code production aspect. This practice also focusses the designers attention on the source code rather than the design when tracking down errors, thereby treating specification errors as development errors. A side-effect of this is that the designer is often tempted to change the source code, compile and test this, and then attempt to reflect the changes upon the original specification.

JSP diagrammers are often thought of as being a convenient method of producing code. They are, but to think of them solely in these terms is holding back the further development of these tools. They are currently analogous to primitive 3GL compilers which produce assembly code rather than an executable program. A JSP diagrammer which generates code should be considered a JSP compiler, and as such the source submitted to this compiler should be executable, in this case - the JSP specification.

A method of execution provided by PROBENCH is called a TRACE facility. This allows the validation of a JSP diagram by generating extra source code, during the code generation phase of JSP development, which writes trace information to a file for later processing. The trace information details which operations are executed and which boxes are visited. The source code is compiled and when executed it produces the trace file, this file is then used by the diagrammer to display a post-mortem run of the diagram graphically. Whilst not being the ideal way to validate, debug and test JSP diagrams it is far better than examining the generated source code.

An alternative presented by researchers [WARHURST & FLYNN 90], is the animation of a JSP diagram. Their aim is too allow a greater degree of specification validation to be carried out by both the analyst and the user. Research carried out by Warhust and Flynn found that "only about 80% of the specification is validated by analysts , with the user validation even less". Their tool uses a subset of english as the operations that can be applied to the diagram and is used as a prototyping tool.

During animation the system interacts with the user via a series of input screens to dictate how the animation will proceed. This allows the quick mock up of a version of the system to be revealed to the users at any early stage of development. Whether or not this is a useful method of presenting JSD specifications, taken to a computable level, to the users remains to be seen. But it has to be better than static presentation, and would certainly be useful to the analyst as an aid to validation. However at a later stage in the development the analyst may wish to validate the JSP specification in terms of full functionality rather than as a prototype. A tool based mechanism for this should also be provided.

5 Tool Requirements

The initial criteria for a tool is that it must have diagramming facilities. These would be used to create the diagram, display it and to highlight the the execution route through the diagram.

This diagrammer must enforce the creation of correct PSDs, using the rules set out for JSP design.

An integrated environment should be provided for the creation of the diagram and the edition of operations, conditions, and quits. The tool should have facilities for handling backtracking and inversion. Code generation facilities should also be provided for the finished code. If possible the diagram should be turned directly into an executable file.

The initial criteria above, simply describes the JSP design tools currently available. The second criteria, an interpreter module, is not provided by any of the current tools.

Figure 2

The Interpreter will take the diagram and the source for the operations, conditions and quits; check for errors; and then execute.

At its most basic level, the executor should syntax check the operations before it begins and then graphically travese the diagram; highlighting each box as it is reached, and each operator & condition as they are evaluated.

More sophisticated systems would allow breakpoints on boxes & operations, monitoring of specific variables, interogation and manipulation of program variables. In short, the basic features associated with a source level debugger - supplied as standard with most modern compilers.

This implies a batch oriented approash to the process of JSP. With a more integrated system, operations could be syntax checked and parsed on-line - just as the diagrammer will not allow incorrect structures to be created, the editors should not allow incorrect source to be input.

A windowing environment of some sort is essential for the creation, editing and execution of the specifications, but these types of environments are normally associated with JSP tools.

As can be seen from figure 2, the roll required is essentially an improved diagrammer and as such could be created by modifying any existing JSP design tool.

There are obvious difficulties associated with the creation of a JSP *execution* tool, rather than a design tool, and these are involved with the creation of a different interpreter for each source language in use. Obviously this is a large task, but in principal it is no different to having a different code generation module for each source code handled by the diagrammer. The benefit of the interpreter is that it can be quite simply converted to a compiler for the source language.

By using an interpreter the specification could be checked for syntactic correctness without generating any code, and once correct the specification could be animated on-line to determine its functional (semantic) validity.

In this way the process of JSP specificaton refinement could be carried out without the need for the generation of incorrect source code thereby removing the problems associated with the refining of JSP generated code.

6 Benefits of JSP Specification Execution

The benefits to the development cycle of JSP specified programs can be seen by comparing Figures 2 3.

FIGURE 3

By the removal of the code generation aspect of specification validation then the temptation to 'fix' the generated source code rather than the design from which it came is removed. There is also no need to attempt to understand the code produced by the generation procedure, simply the specification diagram which is familiar to the designer.

Emphasis of JSP is placed back on design and specification that can become the program, rather than as a guideline for producing a program.

Automatic generation of test data can be approached in a number of different ways, random test data generation, wide coverage testing - generating a large amount of test data from simple rules, and test path generation.

Test path generation is useful to help minimise the amount of test cases supplied, and to ensure that the program is tested as thoroughly as possible.

The use of a specification execution tool opens up the possibilty of combining several approaches. Test path generation is a well researched topic in relation to JSP [ROB's PROJECT], and this is aided when the diagram is held in an automated tool. The difference between a specification execution tool and a JSP diagrammer is that the execution tool has slightly more semantic knowledge about the specification than the diagrammer. By this I mean that the executer also knows about the program identifiers that exist at the operations level and conditions level of the diagram. So if a test data generation tool was to be combined with the executing tool then not only the paths could be generated,

but also the data needed to initiate those paths. Extra information such as which other identifiers are altered and used in the course of the path could be identified at the same time. All this data would then be subject to the analysts scrutiny but it would provide a far greater coverage of the testing process than has been available at the moment.

Automatic generation of test data becomes easier since the interpreter can work through the specification determining test paths, considering what data is necessary to walk those paths and could even produce symbolic executions of those paths.

The test data produced can be quickly fed back into the specification to see how it reacts.

Dynamic testing becomes more complete, traditionally dynamic testing only automates the execution of the test data rather than the creation of the test data. This is because any test data generated would be done based on the source code and so would not test for errors since the test data itself would be based upon the erroneus source code.

With the test data generation being done from the specification it removes any logical errors added by the programmer. And the item being tested is the specification. The test data can be generated because the paths through the program can be mapped onto the data necessary to generate those paths. Not just symbolically but the actual data itself.

Implications also arise for training new staff in the use of JSP, the emphasis of a training course would be less on the control constructs of a computer language such as COBOL with its many sections and paragraphs, but on the simpler constructs of JSP notation.

The COBOL language features would obviously have to be covered at a later date in order to provide a complete education, but there is no need to overload the novice at the start of his training by presenting him with a means of specifying his programs and then with another set of notation to check that his specifications are correct.

Although the specification itself may have been tested and deemed correct, the resulting program may still need to be debugged, the benefit of the executable specification system is that there is no need to have defensive code embedded in the final system to aid with debbugging. For example, the use of DISPLAY statements in COBOL code becomes redundant when the tester can see exactly where on the diagram the program is. The use of dedicated testing code can cause problems; for example some of the errors in a system may be due to the debug code itself, similarly errors may be introduced with the removal of the debug code. These problems are circumnavigated by avoiding the need for such devices.

Maintenance requires clear and accurate documentation in order to be successful, particularly when the person maintaining a system is not the same person who wrote it.

Maintenance of old projects is recognised as a difficult task. JSP is useful

during maintenance because only one document needs to be maintained, and with a means of executing JSP specifications then this becomes a little easier. The main benefits provided are those that would be associated with a JSP diagrammer at any rate. These being; up to date documentation, a simple mechanism for changing the program, all the information about the program being easily accessible. The extra advantages relate to the ease of testing the modified code, the test data could be run through the program under programmer control.

Execution under an interpreter is obviously slower than the associated compiled code. However the intention is to produce a system capable of allowing the developed specification to be tested and validated under analyst control easily and naturally. The final system is not expected to be run within the interpreted environment, just as a prototype is not expected to be the final version of a system. Code generation facilities would allow the move to be made from executable specification to the compiled version without any problems.

This obviously gives rise to the consideration of the JSP specification as a prototype. Dependant upon the definition of prototype used then this can be applied to any executable specification mechanism or possibly any specification mechanism.

Prototype is derived from the greek "protos" meaning first and "typos" meaning a type. The Collins National Dictionary of 1966 classes a prototype as the 'original or model from which anything is copied'. The Shorter Oxford Dictionary uses a similar definition ' the first or primary type of anything; a pattern, model, stadard, exemplar, archetype'. Obviously a specification is the type or model of a system and as such is a prototype, in the dictionary sense. In computing circles a prototype is generally accepted to be executable, and this tool will allow the execution of a JSP specification and could really be classed as a prototyping tool.

7 Future

JSP tools, although having existed for many years are still in their infancy. Their current state is analogous to early compilers whereby source is converted into assembly language for further processing, although JSP tools are rarely even as advanced as this and are more akin to macro processors, but since they come in integrated environments with graphical displays and editors they are percieved as being more advanced tools than they really are. JSP tools have the potential to be far more than simple diagrammers, as they currently are. They could be used to provide a vast array of powerful software development aids, but before this can happen, they must first become executable.

References

[YONEZAKI 89] Naoki Yonezaki, *Natural language interface for requirements specification* , in Japanese Perspectives in Software Engineering, edited by Yoshihiro Matsumoto and Yutaka Ohno, published by Addison-Wesley, 1989

[HEIDORN 76] G. E. Heidorn, *Automatic Programming through natural language dialogue: a survey*, in Readings in Artificial Intelligence and Software Engineering, Edited by Charles Rich and Richard C. Waters, published by Morgan Kaufman Publishers, 1986

[ORMSBY 91] Andrew Ormsby "Object Oriented Design Methods," in *Object-Oriented Languages, Systems and Applications* Edited by Gordon Blair, John Gallagher, David Hutchison and Doug Shepherd, published by PITMAN 1991.

[C-C D POO 91] C-C D Poo, "Adapting and using JSD modelling technique as front-end to object-oriented systems development," *Information and software technology* vol 33 no 7 September 1991

[ABBOT 80] R, Abbot., "Report on teaching Ada," *Science Applications Innc., report SAI-81-312WA,* December 1980

[SPENCE 91] I.T.A Spence and B.N. Carey, *Customers do not want frozen specifications*, Software Engineering Journal, July 1991

[VONK 90] Roland Vonk, *Prototyping: The effective use of CASE Technology*, Prentice Hall, 1990

[BOOCH 83] Grady Booch, *Object oriented development*, IEEE transactions on software engineering, volume se-12, number 2, february 1986, pages 211-221

[PARTSCH 90] Helmut A. Partsch, *Specification and Transformation of Programs: a formal approach to software development*, Springer-Verlag, 1990

[LISKOV & BERZINS 79] Barbara H. Liskov and Valdis Berzins, An Appraisal of Program Specifications, in *Software Specification Techniques*, edited by N. Gehani and A.D. McGettrick, 1986,

[LISKOV & GUTTEY] Barbara Liskov and John Guttey, *Abstraction and Specification in Program Development*, MIT Press, 1986

[KAISER et al] Gail E. Kaiser and David Garlan, Synthesising Programming Environments form Reusable Features, in *Software Reusability Volume II Applications and Experience*, Edited by Ted J. Biggerstaff, Alan J. Perlis.

[BALZER et al] R. Balzer and N Goldman, "Principles of Good Software Secification and their Implications for Specification Languages" in*Software Specification Techniques*, edited by N. Gehani and A.D. McGettrick, 1986

[MEILIR 88] Page-Jones Meilir, *Practical Guide To Structured Systems Design*, Prentice-hall, 1988

[MARTIN 85] James Martin, *Sytem Design from Provably Correct Constructs*, 1985, Prentice-Hall

[WARHURST & FLYNN 90] R. Warhurst and D. Flynn, "Validating JSD specifications by executing them", *Information and software technology*, vol 32 no 9 November 1990

[RATCLIFF 87] B. Ratcliff, *Software Engineering: Principles and Methods*, 1987, Blackwell

[SOMMERVILLE 85] I. Sommerville, *Software Engineering*, Addison-wesley, 1985

[SWARTOUT & BALZER] William Swartout and Robert Balzer, "On the Inevitable Intertwining of Specification and Implementation", in*Software Specification Techniques*, edited by N. Gehani and A.D. McGettrick, 1986

[DROMEY 89] Geoff Dromey, *The development of Programs from Specifications*, 1989, Addison Wesley

[BOOCH 91] Grady Booch, *Object Oriented Design With Applications*, 1991, Benjamin Cummings

[KEMMER 89] Robert G. Kemmer, *PRODIGY - A program development workbench*, M.Phil 1989