JSD VERSION 2 AND AN OVERVIEW OF DEVELOPMENTS IN JSD

J. R. Newport

This paper provides a brief description of the JSD method as it was originally published in 1983[1] followed by a description of the enhancements introduced over the years which were included in version 2 of the method published in 1992[2] and finally it covers some of the recent optional extensions to the 'core' JSD method.

JSD was developed by M.A. Jackson and J.R. Cameron over a period of four years and was first published in 1983[1]. Arguably, this makes it one of the first Object Orientated software development methods. The word 'Arguably' relates mainly to what constitutes the definition of 'Object Orientated' (or Object Oriented). JSD identifies Objects (referred to as Entities within JSD) by time ordering Events (referred to as Actions within JSD) to build up a dynamic data model of the real world. The original version of the JSD method was 'Object Orientated', in that it was identifying objects as part of the method at a time when the more widely used methods were based on functional decomposition or static data analysis. Thirteen years after the JSD method was first published, Object Orientated methods are now more widely used. Over that same period the definition of what constitutes an Object Orientated software development method has been extended from simply being a method that is orientated around objects to one that includes inheritance, polymorphism, encapsulation and public and private data. Although JSD can be used in a way that allows the inclusion of a considerable degree of inheritance and polymorphism, these features were not specifically stated as being part of either the original nor the latest published version of the method[2]. Various organisations have proposed extensions to JSD to specifically include inheritance and polymorphism but their use has been localised within these organisations and the extensions have never been standardised.

A system specification in both Version 1 and Version 2 of JSD consists of a network of processes called a System Specification Diagram (or SSD). Diagram 1 shows a network of JSD processes connected by Datastreams and State Vector Inspection connections as it would appear in JSD version 1.

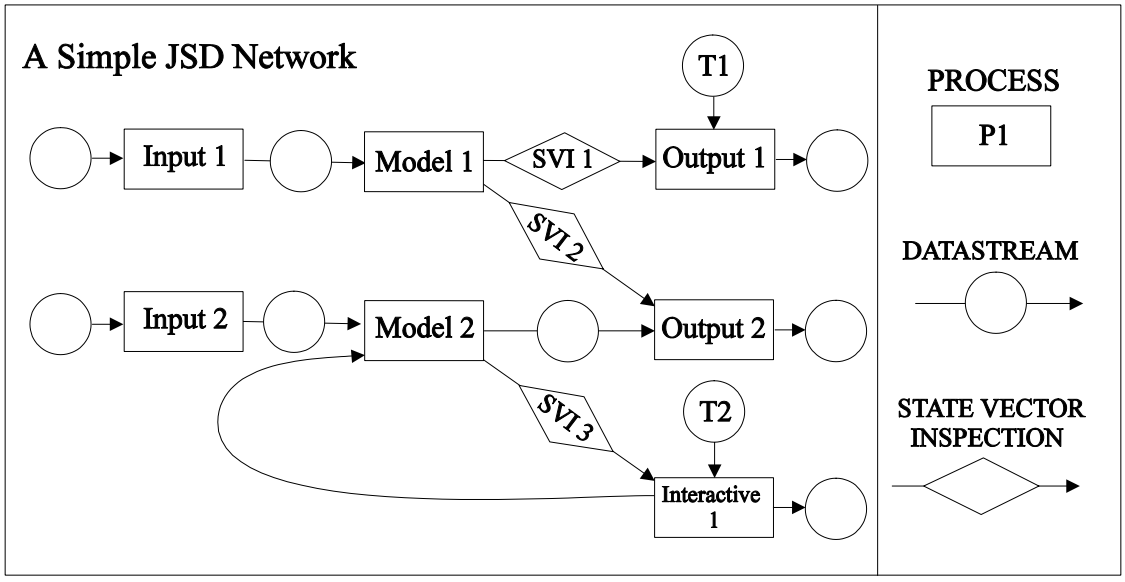

Diagram 1

J.R. Newport is the secretary of the JSD User Group and is a director of Keywood Computer Services Limited, Jordans, Beaconsfield, Bucks., HP9 2TG.

Each process in the network represents a sequential thread of execution and is specified using a structure diagram which, in addition the sequence, selection and iteration constructs, includes the detailed operations expressed in the implementation language to be used. JSD processes are of two main types; Model Processes, which specify a model of the real world, and Function Processes, which are designed using Jackson Structured Programming (JSP) and which feed Actions through to the Model and which then extract and output information as required. Although strongly influenced by Professor Hoare's work on Communicating Sequential Processes[4] (CSP), JSD is not itself a formal method but in common with true formal methods, JSD develops a detailed specification which contains all the information necessary to generate a complete implementation. Although JSD specifications can always be implemented by hand, because of the detail held in the specification it is also possible to fully automate the implementation stage of JSD using an automatic code generator with the resulting implementation suffering little or no loss in run-time performance[5] when compared to a manual implementation.

Version 2 of JSD added Roles, Action Renaming, static data modelling, Conversational and Controlled Datastreams and the option of using State transition diagrams to time order Actions. In version 2, Entities no longer have their own structure diagrams, instead they have one or more Roles and each Role has a structure diagram. JSD version 1 allowed concurrent systems to be specified, with the Actions of each Entity occurring concurrently with the Actions of the other Entities in the system. However, as there was only one structure diagram per Entity and hence one sequential time ordering of Actions for each instance of each Entity, it was not possible to specify concurrent behaviour within a particular instance of an Entity. Taking the example of an army tank in a battlefield simulation where the tank has four Actions: Start, Stop, Load and Fire, a single structure diagram is sufficient to specify a tank which can only fire its gun when stationary. For simplicity, it is assumed, slightly unrealistically, that for the purposes of this example that the gun will always be fired immediately after being loaded. This is shown in Diagram 2.

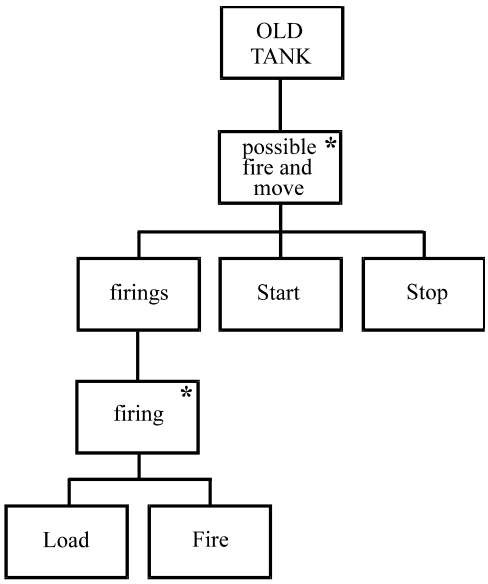

Diagram 2

The time ordering of Actions for 'OLD TANK' shows that 'Load' and 'Fire' can only occur before a 'Start' and after a 'Stop' (the '*' in the top right-hand corner of a box indicates an iteration of zero or more occurrences). However, modern tanks have stabilised guns which can be loaded and fired while on the move as well as when the tank is stationary. The Actions of the gun are time ordered as Load, Fire, Load Fire .. and the movement Actions are ordered as: Start, Stop, Start, Stop .. but these two sequences are completely independent of each other and can be interleaved in any order, providing of course that the time orderings of the gun Actions and the time orderings of the movement Actions are maintained, for example: Load, Start, Stop, Start, Fire, Load, Fire, Stop, Load, Start, Fire, Stop .. . This is because the firing of the gun and the movement of the tank are separate Roles of the tank which can operate concurrently within a single Entity (the tank). This is shown in Diagram 3.

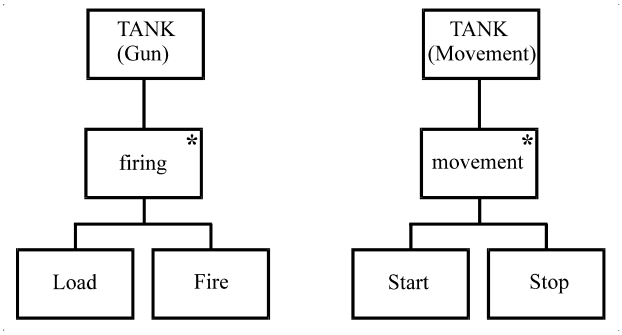

Diagram 3

In the Tank example above, the Action occurrences in the Role structure had the same name as the Base Action but when an Action is common to more than one Role and its effect on the Roles is different then it is useful to rename the Base Action in one or more of the Roles to which the Action is common. 'Action Renaming' can be used simply to rename a Base Action (whose name has been fixed in a requirements document) to a more meaningful name within a Role but Action Renaming has much more useful purposes when applied to Common Actions with asymmetric effects and in specifying specialisations and generalisations. Common Actions synchronise two or more Roles whose other Actions (which are not Common Actions) occur asynchronously. Some Common Actions may have an asymmetric effect on the Roles to which they are common. Returning to the example of a war game simulation, an anti tank missile might always explode if it hits a tank but some tanks may be destroyed by a hit while others are merely damaged, depending on certain criteria within the simulation (type of armour, where hit or angle of impact). A Base Action of 'Hit' may therefore be renamed in a Role of the missile Entity to 'Explode' and in a Role of the Tank Entity to 'Destroy' or 'Damage'. Action Renaming can also be used for generalisations, for example in a system which requires a user to enter a password in order to log on to the system there could be a generalised 'Logged On' Action between 'Log On' and 'Log Off' in a 'Log On' Role. Every Action which can only occur after the User has logged on would be a Common Action between its own Role(s) and the 'Log On' Role. As Common Actions are rejected if they are out of context in any one of their Roles, such Actions would be rejected if the User had not logged on correctly.

Static Data Modelling is a technique which would often be used in JSD to complement the dynamic data Modelling carried out in the JSD Modelling stage. JSD version 2 simply makes it explicit that Static Data Modelling is part of the JSD method.

Two new Datastream types are included in JSD version 2: Conversational Datastreams and Controlled Datastreams. An example JSD version 2 network showing these additional Datastream types is shown in Diagram 4.

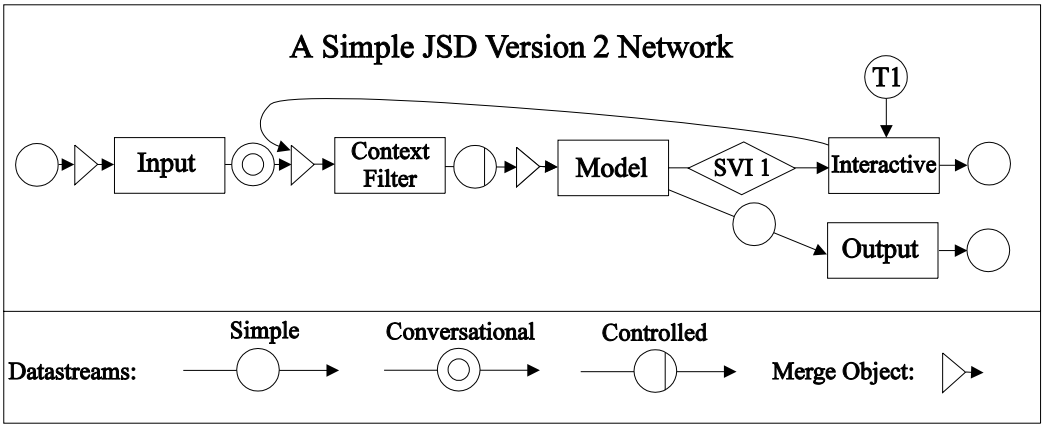

Diagram 4

Conversational Datastreams are simply a short hand graphical notation for a Datastream connection in which a message sent in the direction of the arrow is always followed by a reply message returning in the opposite direction. At implementation time, Conversational Datastreams are typically transformed into procedure calls with both an input and an output parameter. Although a Controlled Datastream is also a short hand graphical notation it encompasses additional operations. A Controlled Datastream consists of a locked interaction in which State Vector Inspections by other processes, on the process being written to, are locked out while a State Vector Inspection is performed in the opposite direction to the arrow, followed by a possible Datastream write operation (conditional on the value of the State Vector inspected) and finally the State Vector is unlocked and made available for other processes to inspect. The Merge objects, represented as triangles attached to processes are to show which Datastreams are to be 'rough merged' together where 'rough merged' means that there is no fixed interleaving between messages from different Datastreams. Two or more Merges attached to the same process would indicate a 'Fixed Merge', for example specifying that a process must read first one record from Merge A followed by one record from Merge B and so on. However, fixed merges are so rare that some projects miss out the Merge objects and include a statement in the specification to the effect that all Datastreams are rough merged directly into the processes.

One of the criticisms of JSD was that many people preferred to use state transition diagrams rather than structure diagrams but JSD version 1 only allowed structure diagrams to be used to specify time orderings. Version 2 allows State transition diagrams to be used as an alternative to structure diagrams to specify the time ordering of Actions. Within JSD, structure diagrams are still the preferred means of specifying time ordering because in most cases structure diagrams show time orderings much more clearly as they are read from left to right with the first Action(s) to occur appearing on the left and the last Action(s) to occur appearing on the right. It is possible to draw state transition diagrams in a similar way so that the Events can be read from left to right but whereas this is an option in State transition diagrams, structure diagrams enforce the left to right rule. Although the inclusion of State transition diagrams in JSD may win the method some supporters, the original reason for including them was that there are a small number of time orderings which can be expressed more succinctly using State transition diagrams.

Although the colloquium for which this paper was prepared is titled "Jackson System Development (JSD) - The Original Object Oriented method ?" it must be said that according to the present day definition, JSD Version 2 is not fully Object Oriented. This is because the present definition encompasses far more than being orientated around objects. JSD was considered revolutionary in the late seventies and early eighties because it did orientate a specification around objects by developing a model of the real world when the mainstream methods were based either on functional decomposition and data flow or on static data analysis. Today the term 'Object Oriented Design' includes features carried over from Object Oriented Programming, including encapsulation, inheritance, polymorphism, and public and private data. While JSD supports some of these features through the use of common Roles, which can be inherited and reused and through the use of Action Renaming to provide a degree of polymorphism, the method would need to be further extended to fully support what are now considered to be essential features of an Object Oriented method. At the last JSD User Group Meeting, held in 1994, the consensus of opinion was that there should not be a JSD version 3. It was considered by the User Group that what JSD version 2 does, it does well and so it was decided that there should be a 'core' method, as defined by JSD version 2 and that users could then add to this as they choose. Many users are happy with JSD just the way it is, some would like support for large systems, others would like to add Task Analysis to define manual procedures in addition to the classic JSD model of the real world while yet others want support for requirements capture, GUI specification and design and the OOD techniques already mentioned. It should also be mentioned that for many users, JSD is too comprehensive and parts of the method are frequently excluded, for example few users specify and merge data structures, as detailed in Jackson Structured Programming (JSP), in order to arrive at the program structures for the JSD Function Processes. But the comprehensive support provided by JSP and by the various stages of JSD are there if they are needed.

Various organisations have already developed their own extensions to the 'core' JSD method. Some have developed variants of JSD for large systems which encapsulate aspects of a system to form subsystems. These variants decompose the system by subject matter (using object orientated decomposition rather than functional decomposition). With suitable extensions to the JSD connection mechanisms, reusable JSD components can be specified before their use is known and then dynamically bound at run-time by sending messages which request that replies are returned to addresses which were not known when the component

was constructed. The classic JSD network may still be used to define the internal structure of such reusable components or it may be used to provide a snapshot or exemplar of a possible network connection at a particular point of execution.

The Defence Research Agency (DRA) have extended JSD to cover Task Analysis, requirements capture, Object Oriented Analysis and HCI design right the way through the software development life cycle to C++ code generation. They have also developed a CASE tool, called GAMBITS, to support this extended version of JSD. This extended method and the GAMBITS CASE tool are described in a paper delivered[6] at the same Colloquium for which this paper was prepared.

If JSD really was the original, or one of the original, Object Oriented Design methods and is also one of the best, then the question must be asked, why aren't more people using JSD? The most likely cause is the lack of JSD CASE tools. There are some very good PC DOS based tools which draw and edit structure diagrams and generate code from individual processes but even though the process internals represent the major part of a JSD System Specification, for most of its lifetime JSD has lacked a user friendly CASE tool to support the complete software development lifecycle. But things are looking up for the method with the work carried out by DRA and my own company has developed a Windows based product to replace the PC DOS based tools, PDF and JSP Tool. Recent developments in software tools mean that CASE tools can themselves be developed more quickly but CASE tools for other methods have more than a head start. The lesson that JSD has had to learn the hard way is that it is the CASE tools that sell the method not the other way around.

References

1. Jackson, M.A.: "System Development", (Prentice-Hall International Inc., London, UK, 1983)

2. Learmonth & Birchett Management Systems: "LBMS Jackson System Development, Version 2.0", (John Wiley & Sons Ltd., Chichester, UK, 1992)

3. Jackson, M.A.: "Principles of Program Design", (Academic Press Inc. (London) Ltd., London, UK, 1975)

4. Hoare, C.A.R.: "Communicating Sequential Processes", (Prentice-Hall International Inc., London, UK, 1985)

5. Newport, J.R.: "To Automate or not to Automate - experiences with JSD", (IEE Colloquium - "Are software development technologies delivering their promise?", 21st March 1995).

6. Powell, D.: "GAMBITS - from requirements capture through to C++ code generation", (IEE Colloquium - "Jackson System Development (JSD) - The Original Object Oriented method ?", 1st February 1996).