The Diagram of Flow: Its Departure from Software Engineering and Its Return

S.J. Morris[1] and O.C.Z. Gotel[2]

[1] Department of Computing, City University London, UK
sjm@soi.city.ac.uk
[2] Independent Researcher, New York, NY, USA
olly@gotel.net

Abstract. The first diagrammatic notation used in software engineering represented the concept of flow. This paper considers the factors that affected the apparent departure of the flowchart from software engineering practice during the 1970s and 1980s and its subsequent return in the 1990s. A new emphasis on hierarchy (as level of abstraction) and on data structure meant that the general concept of flow was completely superseded, only to re-emerge later as a new duality of control flow and data flow. This reappearance took a variety of forms with varying semantics until its stabilisation in the latest version of the Unified Modeling Language. Flow is there re-instated as a fundamental concept in software engineering although its importance, and that of the activity diagram used to represent it, diminished as a consequence of its becoming just one among a wider set of paradigms for software systems development, each associated with its own diagrams.

Keywords: Activity Diagram, Diagrammatic Notation, Flowchart, History, Representation, Software Engineering.

1 Introduction

Earlier papers [1], [2] traced the history of flow diagrams from their early beginnings as representations of flows in nature and industry, through the introduction by Goldstine and von Neumann of a symbolic representation of sequence as a means of machine control, and through the use of their new version of the flowchart as an essential tool for programming and the automation of applied mathematics during the 1940s and 1950s, to the falling of the flowchart into disrepute as an unwieldy and counterproductive assistant for programming. This paper continues this history by considering the demise of the flowchart and its resurrection in the latest software engineering modelling language in an altered form and less crucial role.

Section 2 contrasts examples of programming practice before and after the initial demise of the flowchart and outlines the many factors leading to attempts to structure what were becoming increasingly large and complex programs and systems design projects. Section 3 considers the restructuring of representations of both program and system design, at first by variants of the flowchart concepts, then by more radical

P. Cox, P. Rodgers, and B. Plimmer (Eds.): Diagrams 2012, LNAI 7352, pp. 256–269, 2012.

hierarchical and data-centric approaches to the entire process. Section 4 deals first with the emergence of the object-oriented paradigm for programming and systems design and then considers two versions of one diagram in the Unified Modeling Language (UML), now the prevailing standard for development support in an object-oriented environment. Examination of this activity diagram shows how the concept of flow remained, although concealed, only to re-emerge quite unmistakably following significant revision of its original version. Section 5 presents general historical and practical conclusions.

2 Replacement of Flow as Program Paradigm

2.1 Contrasting Practices

In 1958 Mike Woodger defined the essential stages of program development in manuscripts that survive in the archive of the National Physical Laboratory (NPL) where he worked from 1945 to 1983. These manuscripts include fully detailed flowcharts of complex algorithms, one of which is reproduced in part in [1] and in full in [2]. Woodger defined programming practice in an era when there was only the particular code of a specific machine available to control it and when the programmer, almost always a mathematician or scientist, was also solely responsible for memory allocation and its manipulation. The flowchart then had a crucial role in the representation of algorithms and hence of programs. The production of a flowchart was an essential step following the analysis of the mathematical problem to be solved and the choice of a suitable procedure. This approach had developed in part because the most frequent tasks then involved complex mathematical tasks only made possible by automated computation at high speed [2].

Another program written by Woodger, not later than 1976, “to allow a user with VDU and keyboard to interrogate an existing stored body of information (about fungi)” survives in its entirety with a full program listing and documentation [3]. He uses it as an example in an article in which he again defines the steps of programming, but now without any mention of flowcharts. He sets out as programming principles the “separation of concerns”, the separation of what is to be achieved from how it is to be achieved, and an “hierarchy of virtual machines” containing objects at an appropriate level of abstraction with their associated operations.

The process that Woodger defines and illustrates makes no use of diagrams. It begins with a “rough planning stage” whose purpose is to define “what is to be done”. The second stage consists of “Further detail: how is this purpose to be achieved”, again principally defined as text. The process then switches from what had become known as a top-down approach to the opposite bottom-up process. The final implementation stage involves direct use of the available “programming language and its implementation ... within the constraints of the OS”.

An eyewitness account of the practices in a major UK company, ICI (Imperial Chemical Industries), during the intervening period indicates that it was a severe shock for many that excellent programs could be written using a completely new

method. This writer, Ken Ratcliff, criticises such practices as leading to "indulgence into flights of bit fiddling, or unnecessary pirouetting on a recently learned technique of pointer arithmetic, cross-sectional definition of arrays, programmer controlled allocation of storage independent of program block structure and other excesses of the do-it-yourself type" [4]. The solution for ICI at the time was the rapid and comprehensive introduction of the data-centric program design principles of Michael Jackson with their emphasis on structure (See Section 3.4 below). This account also indicates how important project and staff management issues had already become.

2.2 Structuring the Unstructured

By the early 1970s, the unstructured complexity of increasingly large programs gave rise to reactions, at first little related to each other, under the titles of structured programming and structured systems development. To place developments in programming techniques and diagrammatic representations in context, the conspicuous causes included a wide variety of factors having different effects:

- The ability to write increasingly complex programs using higher level languages which used the original level of machine code as their basis and translators or compilers to produce code executable on a particular machine;
- The introduction of operating system programs to remove much of the effort from memory allocation, file management and the mechanics of input and output;
- The availability of mass storage devices (e.g., magnetic tape, drum or disk);
- The shift in commercial domains (and in other domains already exploiting rapid mathematical computation) towards mass data transformations and manipulations;
- The general lack of experience in dealing with the problems of scaling;
- The craft nature of programming;
- The indications of a continuing and rapidly expanding demand for machines and programming.

As discussions continued, the characteristics of structured programming coalesced around a number of issues, exemplified by a list published in 1978 by Infotech International [5]: structured analysis and design; structured coding; top-down implementation and testing; Hierarchy, plus Input, Process, Output technique (HIPO); team operations; project support libraries; structured walkthroughs; and project management systems. The latter four issues concern improvements to the organisation of programming personnel and the management of the programming process, and are not associated with any particular diagrammatic forms. It was expected that the solution of the former issues would draw on the increasing amount of material on formal, procedural methods for the design and construction of programs.

There is a clearly perceived switch from sub-division of the total problem into manageable parts to the use of formal criteria and techniques for decomposition and design. In the code itself, elimination of the uncontrolled use of GOTO statements was the benefit gained from the use of only three basic constructs (sequence, iteration and selection), credited to Böhm and Jacopini [6]. Top-down implementation and

testing accepted the principle that software development should proceed from the highest control level modules downwards.

3 Restructuring the Representation of Program and System

3.1 Initial Responses

As they manifest themselves initially in program and systems design techniques, the responses to these issues focused on two issues both in procedures and in graphical representations: hierarchy (as level of abstraction) and data structure. The consequence was the complete supersession of flowcharts as the principal basic program design device (as used by Woodger in the 1950s) and its demotion to, at best, a small low-level supporting role.

These responses appeared in a number ways: alterations to the basic flowchart to accommodate some indication of hierarchy (e.g., Dill, Hopson and Dixon); more radically different representations of nested structures (e.g., Chapin); completely data-centric views of process (e.g., Jackson); and combinations of both data and hierarchical approaches in complete system descriptions (e.g., HIPO).

3.2 Tree Chart

As a preamble to examining these new diagrams it is necessary to first review another, the tree diagram or tree chart (Figure 1) to show the paradigmatic representation of hierarchy and how it came to be interpreted in the context of structured programming. The tree chart represents the results of the parsing of functions, or functional decomposition, as usually credited to Knuth [7].

The execution or control sequence represented by the diagram of Figure 1 cannot be, for example, A, C, J, P, A. The executing machine may or may not invoke the execution of the connected functions on the next lower subordinate level (B, C, D or E) in any sequence and any number of times. The essential proviso is that control always returns to the invoking function or module. Likewise with B and its invocation of F, G and H, and so on. This notion of nesting fundamentally altered program structure and introduced hierarchical levels not present in earlier generation programs, even if they made extensive use of routines and sub-routines. The value of decomposition of this type to overcome problems of size and complexity was a fundamental influence on the development of other diagrammatic representations.

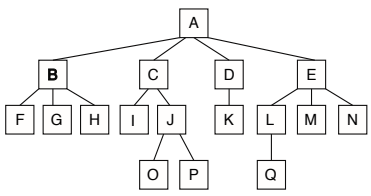

Fig. 1. Hierarchical tree structure for program structure and control

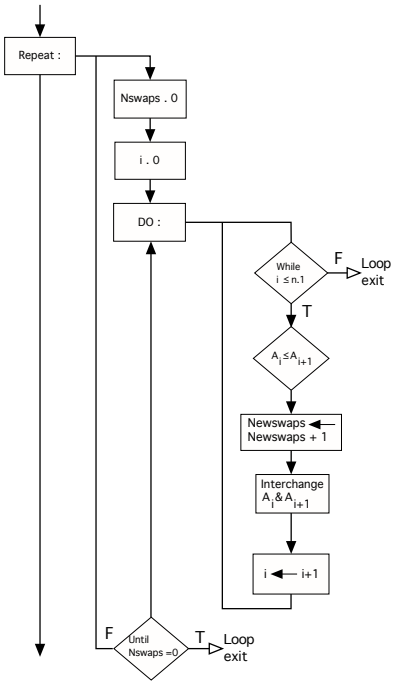

Fig. 2. Flowchart with left-to-right levels of nesting

3.3 Flowcharts Adjusted and Block-Like Variants

The standard ANSI and ISO flowchart [8], [9] imposed no configuration constraints other than a basic vertical sequence from top to bottom plus a reverse parallel path to represent returns to earlier positions in the sequence. Subsidiary horizontal paths served only to connect the single main parallel to its reverse companion(s). Such a structure served well until decomposition demanded some representation of hierarchy. Early attempts to do this, typified by the approach of Dill et al. [10], involved creating a horizontal left-to-right hierarchy, an early version of the graphical trope now called swim lanes. Loop exits now lead not directly back to a point above but to one to the left in a level higher in a hierarchical rather than topographical sense. The diagram shown in Figure 2 represents an exchange sort algorithm that repositions the numbers in an array in ascending order.

An alternative view of nesting, which came to be known as the Chapin chart, sacrificed all direct representation of flow and sequence in order to show nesting and selection. The diagram of Figure 3 [11] represents the same exchange sort algorithm as that described above. This innovative use of a two-dimensional space did not however solve any of the problems of complexity and size in a conclusive manner.

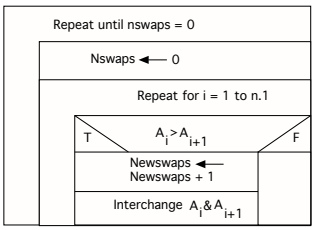

Fig. 3. Internally nested representation of loops in a Chapin chart

3.4 Data-Centric Views

The most radical approach was to switch completely to a program structure not only based on data but also independent of programming language, as exemplified by the initial work of Jackson [12]. The force of its impact comes over clearly in the same account from ICI UK [4]: "(He) casts many existing highly developed techniques to the winds, making a clean start with the explicit principle that program structure should be based on the structure of the data on which the program operates. To facilitate this development he defines a simple but clear notation for the basic components of structure, corresponding to the four constructs of structural coding and called sequence, iteration, selection and the elementary components. Because of the clear structure of stages, and the structure outlined, the design proceeds without the help (or rather hinderance) of flow charts."

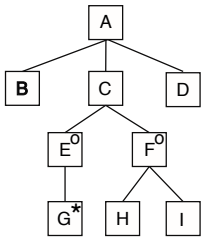

Fig. 4. Data structure in JSD notation

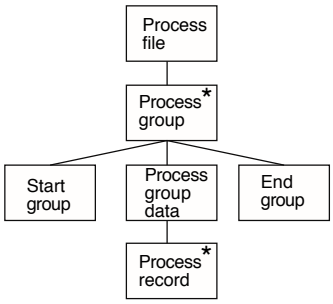

Fig. 5. Data-centric program structure

Figures 4 and 5 show the data and program structures as represented in a contemporary paper of Jackson [13]. In both cases, these represent tree structures in which elements annotated with a '0' are alternatives and those annotated with a '*' may be repeated 1 to n times.

3.5 Whole System Representation - HIPO

The Hierarchy, plus Input, Process, Output (HIPO) technique was intended to describe a whole system in terms of inputs, outputs and constituent, intervening processes. Originally developed at IBM for its own use [14], this takes the separation of concerns and top-down development as fundamental principles. It was not intended as a replacement for older techniques, rather as a means of system description at a level that was not possible to achieve previously when using them.

Figure 6 shows the basic notation used to represent a hierarchy of process components belonging to a main process P (in this case read at the subordinate level from left to right with P_2 following P_1) and the data inputs A and B to process P and its outputs C, D and E.

With the advent of techniques such as HIPO, and the many others that appeared during the 1970s (see Davis [15] for a review), the displacement of the flowchart as an essential programming tool was complete. Development of alternative programming paradigms demanding completely different representations meant a lapse into obscurity for the flowchart *per se* as a programming tool.

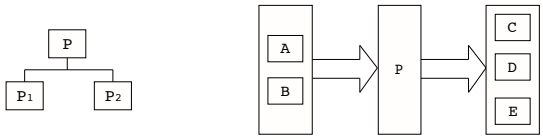

Fig. 6. Hierarchy of components in HIPO (left) Inputs to and Outputs from process P (right)

4 Flow Survives Object-Orientation

4.1 New Object-Oriented Approaches

From the earliest discussions of structured programming [16], basic concepts of object-oriented programming were emerging in the form of discussion of data types and their associated operations. The development of a completely different object-oriented paradigm for programming, derived in part from such notions of abstract data types, meant that a wholly new set of diagrams was created. In addition, a whole new class of diagrams for data and database structuring were emerging, which are outside the scope of this paper.

In a repeat of what had occurred in the 1960s and 1970s, there was again an explosion of diagrammatic forms required to assist the development process and an attempt to clarify complexity, leading to an effort to create initially a Unified Method

[17] and then a Unified Modeling Language (UML) [18] for object-oriented development. The UML has subsequently gone through many revisions, the latest major revision being Version 2.4 [19].

Flow, in the sense of a sequence of activities needed to achieve a particular goal, had not disappeared, nor had the need to represent notions of the selection of alternative routes and the possible repetition of segments in such sequences. Hence the many various uses of flowcharts continued outside the restricted domain of algorithmic expression and computation.

Within the evolution of the UML the role of flow has gone through three phases. In the first phase, flow played no recognised role at all in the semantics of the closest relative of the flowchart, known from the outset as the activity diagram. In the second phase, object flow became significant. In the third phase, object flow and the original concept of control flow form a new duality and mark the re-emergence of a full role for flow in the UML.

During the first and second phases, the principal underlying paradigm for behaviour was the finite state machine (FSM): "a hypothetical machine that can be in only one of a given number of states at any specific time. In response to an input, the machine generates an output and changes state. Both the output and the next state are purely functions of the current state and the input". [15]. Its particular development in the form of the state machines of Harel emphasises the importance of "the hierarchical decomposition of finite state machines and a mechanism for communication between concurrent finite state machines" [15]. This variant of the FSM provided the initial formal semantic basis for the initial activity diagram while another variant, the Petri net [20], contributed to its visual syntax. In the third phase, this essentially transitional view of behaviour was displaced in the activity diagram by the sequential view always implicit in the visual syntax. In the following sub-sections we examine the details of UML development and show how this occurred.

4.2 Initial Version of the UML Activity Diagram

When what was to became known as the Unified Modeling Language (UML) was proposed in 1995 [17], the software engineering community was presented with seven different types of diagram for specifying and designing a software-intensive system that would be implemented in an object-oriented manner (i.e., class, use case, message trace, object message, state, module and platform diagrams). Notably absent was any diagram specifically claiming to model the concept of flow.

The addendum to the initial version of the UML, known as Version 0.91 [18], remarked that: "Sometimes it is useful to show the work involved in performing an operation by an object." Activity diagrams were introduced to fill this gap in the emerging UML and therefore to provide a way to show the method for implementing an operation visually (i.e., the steps that occur in the procedural implementation of an operation). The activity diagram was not labelled as a flow diagram or flowchart; rather, it was described as: "a special kind of state machine that describes the implementation of an operation in terms of its sub-operations." These sub-operations were essentially the procedural activities internal to the object owning the activity

diagram and referred to as activity states. The semantics of the initial activity diagram were explicitly grounded in those of state machines.

Examining the exemplar activity diagram in this addendum document, shown in Figure 7, reveals a visual notation with distinct graphical icons for the following:

- ***Activity states*** - Rounded rectangles that contain the name of a single activity inside. These model the individual steps (i.e., activities) in the implementation of an operation.
- ***Transitions*** - Solid lines, with a single directional arrow on one end. These model the sequencing between the activity states (i.e., the flow of control). The transitions are implicitly triggered by the completion of the preceding activity state.
- ***Synchronization bars*** - Solid thick horizontal lines. These model either the initiation or merging of concurrent control, the former when there are multiple arrows leaving a synchronization bar and the latter when there are multiple arrows entering it.
- ***Dummy nodes*** - Small hollow circles that chain guard conditions to model complex conditions on a transition.

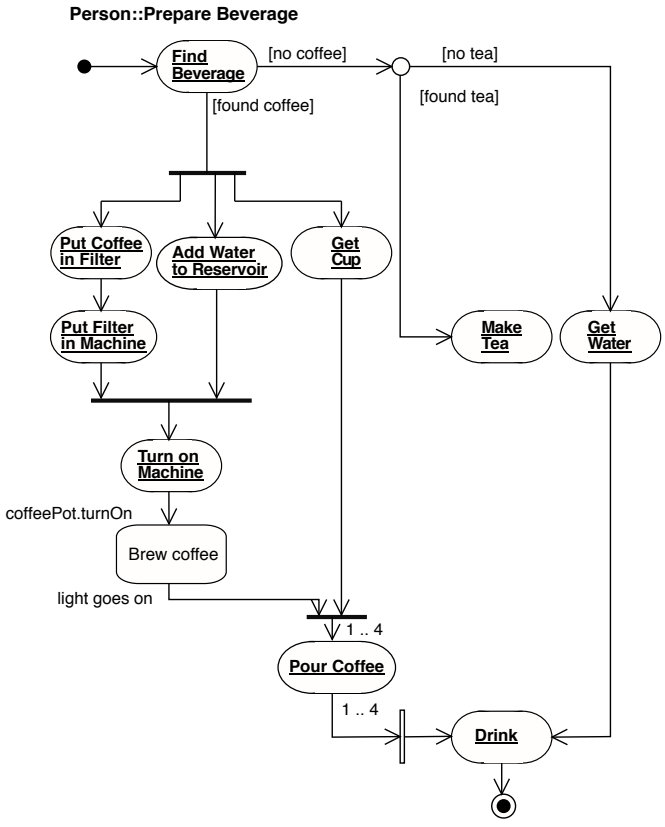

Fig. 7. Earliest version of the UML activity diagram [18]

- ***Start state*** - Small solid circle modelling the starting activity state in the implementation of an operation.
- ***End state*** - Small bulls-eye circle modelling the ending activity state in the implementation of an operation.

The activity diagram is read from the start state to the end state, and the exemplar diagram in the addendum document shows the steps in the implementation of the operation flowing down the page. The accompanying text also refers to the ability to model interrupts to the normal procedural flow of control and to model wait states in order to show operations external to the object but essential to the completion of the modelled operation, though no additional graphical icons are provided.

4.3 A Flow Diagram by Any other Name?

While the first appearance of the UML activity diagram clearly does depict the flow of control, it does not claim to use any of the standard flowcharting icons. Instead, the activity diagram claims to borrow from the notation used by the early UML state machine diagrams for representing its activity states, transitions, start state and end state. It also adopts the bars of the Petri net notation for modelling synchronisation [20], although Petri nets were not part of the syntax or semantics of the early UML.

However, in comparing the earliest incarnation of the activity diagram and the flowchart standards it is clear that there are considerable similarities between the concepts they model. Both activity diagrams and flowcharts model:

- Where the flow of control starts from and where the flow of control ends, though using different icons;
- Discrete processing steps, the processing being decomposed in the activity diagram into activities and represented in a homogeneous way using rounded rectangle icons, while in the flowchart processing types are differentiated with generic processing steps represented using rectangles, input/output represented using parallelograms, and preparatory work represented using hexagons;
- The flow of control, both using solid lines with arrow heads, control passing to the icon to which the arrow points;
- Parallel processing and concurrent control synchronisation, with activity diagrams using a thick single bar icon and flowcharts using two parallel horizontal lines;
- Interrupts in the flow of control;
- Processing steps running down the page, with the ability to express a sequence of operations, concurrent operations and branches in the flow of control, thus both having the potential to express the algorithms of program design and workflow;
- Nested diagrams.

While the activity diagram was described as a specialised form of state machine diagram in Version 0.9, it was evidently to be used to model a progression (or flow), something that was more the remit of flowcharts than state machine diagrams. Moreover, there was no explicit notion of progression in state machine semantics. Given the similarity in the underlying concepts which the icons of both the early

activity diagrams and standard flowcharts were able to model, one could argue that it was the denigrated status of the flowchart, along with the paradigm shift from procedural to object-oriented development, that led to the redesign and renaming of a diagram intended to model familiar concepts.

4.4 Incorporating More Features From Flowcharts

With the release of Version 1.0 of the UML in 1997 [21], the activity diagram was firmly positioned as one of the core diagram types for modelling behaviour. The activity diagram was expanded in scope and intended to be attached not only to the implementation of operations, but also to classes and use cases, focusing on the flows driven by internal processing (i.e., the procedural flow of control), and not on external events. With a statement suggesting the "use (of) ordinary state diagrams in situations where asynchronous events occur" the activity diagram became the *de facto* choice for modelling all forms of synchronous behaviour.

In Version 1.0, the activity state was now relabelled as action state and the transitions were elaborated to add optional actions in addition to guards. The original icon for the dummy node, the small hollow circle, was replaced with a diamond shaped icon to represent a decision point, the exact same symbol that had long been used in traditional standard flowcharts. The action states could further be organised into swim lanes, depicted by solid vertical lines running down the length of a page, to allocate the actions to the objects responsible for their undertaking, reflecting the growing popularity of the use of activity diagrams for business modelling. Such swim lanes also reflect the structure of the much older flowchart variant shown in Figure 2.

In addition to the control flow indicated by the solid line transitions, object flow also began to be modelled using dashed lines to show those objects responsible for performing an action and those objects whose values are determined by an action. This was effectively the modelling of object message passing. The state of an object at any one time could further be described within a rectangular box. Additional icons to model the sending and receipt of signals were also added to the visual notation as its catalogue of icons began to grow. These were referred to as control icons from Version 1.1 onwards and were intended to elaborate the information that could be specified on a transition.

Thus the activity diagram steadily became more elaborate with each successive revision of the UML. By Version 1.4.2 in 2004 (and an ISO standard in 2005 [22]) the main distinction had become a focus on the modelling of nested structures, with both action states and sub-activity states being specified, along with new icons to show this nesting. The diamond decision icon was also now being used to merge decision branches, in addition to simply initiating them, introducing junction pseudo states. The control icons also continued to multiply, in particular increasing support for modelling concurrency. Throughout all these specification revisions, activity diagrams remained defined as a specialised form of state machine diagram while, according to anecdotal evidence, practitioners and tool vendors were coming to refer to them as the flowchart of the UML.

4.5 Acceptance of Flow in the UML

When UML Version 2.0 was released in 2005, activity diagrams were completely reformulated based on the Petri net semantics of token flow. “By flow, we mean that the execution of one node affects, and is affected by, the execution of other nodes, and such dependencies are represented by edges in the activity diagram.” [23]. The change in the base semantics was made to increase the number of flows that could be modelled by the UML and perhaps to reflect increased interest in business process and workflow modelling for systems development.

Version 2.0 specifically delineated the concepts of action (“the fundamental unit of behaviour specification”) and activity, which provides the conditions and sequencing information for coordinating the lower-level behaviours. The primary modelling artefacts of the earlier activity diagram were renamed as activity nodes and activity edges. Distinct icons were provided to indicate the various types of node (e.g., action nodes with a rounded rectangle icon, control nodes with a regular rectangle icon and five types of control node each with their individual associated icons). Concepts were introduced to depict iteration and data storage, both common to flowcharting, and a whole series of icons were introduced to model the concept of containment in activity diagrams. Moreover, the traditional flowchart icons for collation and summing junction were now being used to model accept event actions and final flow nodes.

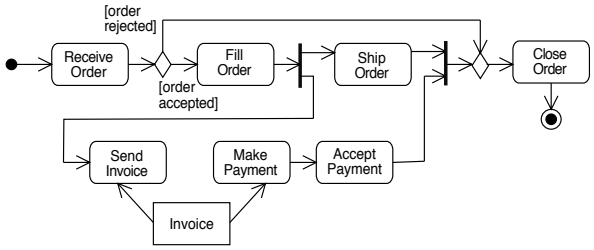

Fig. 8. Example of a UML activity diagram from UML Version 2.4 [19]

With UML Version 2.0, not only was the modelling of control flow and object flow made quite explicit for the first time, but an unequivocal way to model data and information flow was also provided. The latest version [19] defines an object flow as "an activity edge that can have objects or data passing along it … which models the flow of values to and from object nodes" and defines a control flow as "an edge that starts an activity node after the previous one is finished … objects and data cannot pass a control flow edge". The consequence of this change is a diagram that represents, in both a semantic sense and in visual appearance, much of the essence of the original flowchart. Figure 8, extracted from Figure 12.35 of the UML Superstructure Specification Version 2.4 [19], provides an illustration.

As the UML continues to evolve the popularity of activity diagrams in industrial practice is in no doubt, evidenced by its prominence in leading commercial tools. These tools also show, however, that the status of flow has altered drastically. The taxonomy of diagrams provided in UML Version 2.4 includes seven structure

diagrams (i.e., class, component, object, composite structure, deployment, package and profile diagrams) and four behaviour diagrams (i.e., activity, use case, state machine and interaction diagrams, for which there are four variants). While it is quite clear that the concept of flow has re-emerged and that its value in the specification, design and development of software systems has unequalled longevity, it has completely lost the supremacy that it initially acquired and maintained for so long.

5 Conclusions

Understanding the concept of flow remains basic to software engineering. Data flow has joined control flow to form a pair of concepts fundamental to the understanding of programs and the design of software systems. The history of the representation of these concepts shows how diagrams can reveal shifts of technology and changes in the manner in which it is exploited. The succession of flow diagrams created and used prior to the reinvention of Goldstine and von Neumann shows their importance to industrial processes. The new form of flowchart invented to represent the flow of control in the earliest automated computational machines then come to dominate programming practice for more than two decades. Its use also spread to every field where the flow of a sequence of decisions would benefit from being shown in a simple diagram.

While this vernacular use continued, to the extent of misuse and caricature, the original diagram acquired the status of an international standard but fell into disrepute as a programming tool. The complexities of programs and the necessity to represent other concepts, in particular data structure and other views of machine behaviour, required the introduction of alternative concepts and diagrams and their promotion. As a consequence, any assessment of the flowchart made now (if it is given at all), emphasises its importance only outside the specialist field of software engineering and programming practice.

Examination of contemporary practice and its conceptual support shows, however, that flow and flow representations remain firmly entrenched albeit in a new role. The history of diagrammatic notations has in this case revealed the continuing significance of an important concept with the possible consequence of improving understanding and practice. Further work will examine the introduction, development and use of other concepts and diagrams that have been central to the history and progress of software engineering.

References

1. Morris, S.J., Gotel, O.C.Z.: Flow Diagrams: Rise and Fall of the First Software Engineering Notation. In: Barker-Plummer, D., Cox, R., Swoboda, N. (eds.) Diagrams 2006. LNCS (LNAI), vol. 4045, pp. 130–144. Springer, Heidelberg (2006)
2. Morris, S., Gotel, O.: The role of flow charts in the early automation of applied mathematics. BSHM Bulletin. Journal of the British Society for the History of Mathematics 26(1) (2011)

3. Woodger, M.: The aims of structured programming. In: Structured Programming. Infotech State of the Art Report. Infotech International Limited, Maidenhead (1976)
4. Infotech Management Report Structured Programming Practice and Experience. Vol. 1: Management Guide to Techniques and Implementation Vol. 2: Management Report on Implementation Practice. Infotech International Limited, Maidenhead (1978)
5. Infotech International Survey Structured Programming Practice and Experience. Vol. 1: Overview of Structured Programming Vol. 2: Structured Programming Methodologies and Techniques Vol. 3: International Survey and Analysis of User Experience. Infotech International Limited, Maidenhead (1978)
6. Böhm, C., Jacopini, G.: Flow diagrams, Turing machines and languages with only two formation rules. Communications of the ACM 9(5) (May 1966)
7. Knuth, D.E.: The art of computer programming - fundamental algorithms. Addison Wesley, Reading (1968)
8. Chapin, N.: Flowcharting with the ANSI standard. A tutorial. Computing Surveys 2(2), 119–146 (1970)
9. International Organization for Standardization. Information Processing - Flowchart symbols. ISO 1028 (1973)
10. Dill, J.M., Hopson, R.W., Dixon, D.F.: Design and documentation standards. Brown University, Providence (1975)
11. Nassi, I., Shneiderman, B.: Flowchart techniques for structured programming. SIGPLAN Notices 8(8) (August 1973)
12. Jackson, M.A.: Principles of Program Design. Academic Press, Orlando (1975)
13. Jackson, M.A.: Data structures as a basis for program design. In: Structured Programming. Infotech State of the Art Report. Infotech International Limited, Maidenhead (1976)
14. HIPO - A Design Aid and Documentation Tool. Poughkeepsie, NY, IBM Corporation, Form SR20 - 9413 (1973)
15. Davis, A.M.: Software Requirements Analysis and Specification. Prentice-Hall International, Englewood Cliffs (1990)
16. Dahl, O.-J., Dikstra, E.W., Hoare, C.A.R.: Structured Programming. Academic Press, London (1972)
17. Unified Method V0.8. Rational Software Corporation, Santa Clara (October 1995)
18. Booch, G., Jacobson, I., Rumbaugh, J.: The Unified Modeling Language for Object-Oriented Development. Documentation Set Version 0.91 Addendum. Rational Software Corporation, Santa Clara (1996)
19. Object Management Group. OMG Unified Modeling Language™ (OMG UML), Version 2.4 (January 2011), `http://www.omg.org/spec/UML/2.4/` (accessed July 26, 2011)
20. Peterson, J.: Petri Nets. ACM Computing Surveys 9(3) (September 1977)
21. Unified Modeling Language V1.0. Rational Software Corporation, Santa Clara (January 13, 1997)
22. International Organization for Standardization. Open Distributed Processing - Unified Modeling Language (UML) Version 1.4.2. ISO/IEC 19501 (2005)
23. Unified Modeling Language Version 2.0. Object Management Group (August 2005)