

# When Are Methods Complementary?

Richard F. Paige

*Department of Computer Science, York University, 4700 Keele St., Toronto, Ontario  
M3J 1P3, Canada. paige@cs.yorku.ca*

**Keywords:** Software development methods; Complementarity; Method integration;  
Software process; Specification language.

---

## Abstract

We address the issue of when software development methods are *complementary*, i.e., determining when a method is capable of a task that another method cannot perform. Our intent is to examine complementarity in order to help determine when to carry out *method integration*. We propose some factors for method complementarity, and suggest that context-dependent criteria, such as real-world domain and non-functional development requirements, may have a significant impact on method complementarity.

---

## 1 Introduction

Software development methods have been the subject of much recent and not-so-recent research. The past twenty-five years have seen the advent of many different styles and classes of methods: structured methods, mathematical methods, object-oriented methods, methods for the development of parallel or distributed systems, and so on.

A standard definition of a method is that it is a systematic way of accomplishing a specific task. A definition of a software development method, which we hope is not too controversial, is as follows.

**Definition 1** *A method for software development consists of a collection of notations, and a process that applies the notations (for specification, transformation, and analysis) in accomplishing the task of software development.*

It has been suggested that no single method for software development is a panacea [17]. That is, there is no individual method that will suffice in meeting all of the functional requirements (requirements on system functionality) and non-functional

requirements (requirements on the development process itself, such as time constraints, cost, and other properties not related to system functionality) in every software development. Technical and philosophical reasons for this are discussed elsewhere [17,19,27,32], but one significant rationale is because of the increasing complexity of the software systems we want to build.

A variety of approaches have been suggested for dealing with limitations of methods. One approach, of recent interest, is that of *method integration*.

**Definition 2 Method integration** *is the process of combining two or more methods to form a new method which is more useful than any of the separate methods.*

Method integration has been studied in a number of contexts: combining formal and informal methods [11,19,25,28], and combining multiple formal methods [4,23,32]. Method integration can involve the definition of relationships between the processes of the methods being combined. It may also involve constructing formal definitions of the meaning of compositions of specifications written in different notations. Method integration can also be carried out via the linking of individual tools that support separate methods, e.g., theorem provers and computer-aided software engineering (CASE) tools [19].

An important issue that is at the heart of method integration is *when to carry it out*. It is this topic we intend to discuss in this paper.

### *1.1 When should methods be integrated?*

A number of different method integrations have been presented in the literature, e.g., [4,19,28]. Typically, each integration has been carried out in order to address specific limitations in one of the methods under study. So as to be able to justify the expense of carrying out method integration, the methods being combined must be *complementary* in some way.

*Webster's Dictionary* [31] defines complementarity as “serving to fill out, or complete”. We suggest that two software development methods are complementary if there is something that can be done with one that cannot be done with the other. This definition allows for several different kinds of complementarity, as we shall see.

The requirement to integrate methods that are complementary manifests itself in the examples of method integration seen in the literature so far: the combinations of formal and informal notations and methods like Structured Analysis (SA) [7] and Z [25], Shlaer-Mellor and Z [12], and the examples in the survey paper [28]; the combinations of multiple informal methods, e.g., an object-oriented and data flow-based method in [27]; or the application of multiple formal methods to developing

an air-traffic control system in [11]. What can be inferred by these and other method integration examples is that:

- method complementarity can be defined in terms of notations *or* processes; **and**
- complementarity can depend on the task to which the methods are to be applied, or on the context in which the methods will be used.

In other words, there are many forms of method complementarity, some of which may be context-dependent. And in this paper, we suggest an initial list of ways in which developers may find methods to be complementary.

It is not our goal to address the issue of how to carry out method integration. This has been the topic of research presented elsewhere, e.g., [11,19,22], and we encourage the reader to consult the references for techniques to carry out such tasks.

## 1.2 Method compatibility

Method complementarity is closely related to method *compatibility*. We suggest that two methods are compatible when the methods can be used together or as replacements. Under this definition, for example, a method that uses  $Z$  (e.g., the  $Z$  ‘established strategy’ [29]) is compatible with BON [30]; see [24] for an example of an integration of such methods. Methods may be compatible in several different ways. Two methods may be compatible if their notations are equally expressive, or if the notations present the same view (e.g., control or data) of a system. Two methods may also be compatible if they have the same process, or if the process of one method encompasses that of the other. Two methods may be compatible if the tools of one method can be used to support a second method.

We do not concern ourselves further with method compatibility in this short paper. We suggest that method complementarity is a more important issue than compatibility. We must have a clear, justifiable rationale for carrying out method integration. With such a rationale, we may be able to justify the (often considerable) expense of integrating incompatible methods.

## 2 The Complementarity of Methods

The constituent parts of a method suggest many forms of method complementarity. Methods combine *notations*, and a *process* which makes use of the notations. For example, Structured Analysis and Design (SA/D) uses data flow diagrams, structure charts, entity-relationship diagrams, and structured text. A traditional process for SA/D has phases for building data and information models via entity-relationship diagrams, and for decomposing and implementing models. A formal method like

B [1] includes a set-theoretic notation and transformation rules, e.g., for data and algorithm refinement.

We start by categorizing forms of method complementarity in terms of notation and process. We also consider some other forms of complementarity that are dependent on the context in which methods are to be used.

For the sake of simplicity, we discuss complementarity involving only two methods. The discussion can be generalized to more than two methods.

## 2.1 Complementary notations

Two software development methods may be complementary in terms of their notations. This complementarity might manifest itself in several specific ways.

- (1) **Semantic expressiveness.** A notation of one method may be able to express semantic concepts that are inexpressible in notation of a second method. For example, with weakest preconditions (*wp*) it is possible to write (angelic) non-deterministic specifications that are inexpressible in the formal notations of Z and predicative programming [13]. *wp* is therefore complementary to Z. As a second example, UML could be considered complementary to data flow diagrams, because the former modeling language can express encapsulation via classes, a facility which is not available with data flow diagrams. So, one might combine methods to acquire the ability to say more with a combination of methods than can be said with one specific method.
- (2) **Syntactic capacity.** A notation of one method may possess a syntax fragment, whereas a notation of a second method may not possess an equivalent syntax. The second notation may be extendible to include the special piece of syntax, but in a setting where methods can be put together, the need for extension is questionable. For example, predicative programming [14] and Z are complementary in terms of a syntax for expressing parallel composition and process communication. Predicative programming has such a combinator, while the Z notation does not. Predicative programming has a syntax for expressing space constraints on systems, while Z does not (though it can be so extended; see [23]). The two notations can therefore be considered complementary.

Zave and Jackson [32] raise an important point about heterogeneous notations in general, when they suggest that in a heterogeneous setting it may be possible to create much simpler, smaller, more specialized notations than is possible when restricted to use of a single notation. Simpler languages, in particular, may make formal notations more straightforward to use or adopt.

- (3) **Syntactic expressiveness.** A notation of one method may possess a piece of syntax that can be used to write smaller, more concise specifications than is possible using a notation of a second method. The second notation may pos-

sess an equivalent or similar piece of syntax—thus, the complementarity is not due to differences in semantic expressiveness or syntax deficiencies—but the first notation is simply more straightforward and convenient to use to write the specifications required. Bowen and Hinchey [6] suggest a specific example of this, with respect to Z and Communicating Sequential Processes (CSP) [15]. At a semantic level, both CSP and Z represent sets. At a syntactic level, CSP can write more concise, simpler specifications of communicating, concurrent systems, while Z can write more concise, simpler specifications of state-based systems. Therefore, the two notations can be seen as complementary in terms of their syntactic expressiveness capabilities. The notations are clearly complementary in other ways as well.

- (4) **Familiarity.** In a specific development context—i.e., for a specific problem, set of developers, non-functional constraints, management process—two notations may be complementary in terms of their respective familiarity to the developers. For example, one subset of the developers may be familiar with data flow diagrams, but unfamiliar with Z. On the other hand, a second subset may be quite familiar with Z, but inexperienced with data flow diagrams. In a method integration context, it may be possible to accommodate both groups of developers, using the notations with which they are the most familiar. This could be beneficial because the project in question may not have the time available to spend on learning new notations or methods.

In such a setting, it may be necessary for there to be at least one developer familiar with both notations, in order to deal with notation incompatibilities.

As our discussion suggests, complementarity in terms of notation can directly imply complementarity in terms of the methods that use the notations. A further example of this arises in instances of integrating formal and informal methods, such as those in the survey paper [28]: such methods are complementary in terms of the notations of each method, since formal methods use formal notations (which are amenable to mathematical manipulation), while informal methods are used to write informal specifications (which cannot be formally manipulated).

Method integration can be justified when complementary notations are involved; many cases of method complementarity will likely arise in terms of the notations that are used, e.g., [4,11,18]. However, if methods are not complementary in terms of notation, they may be complementary in other ways.

## 2.2 *Complementary processes*

Methods have both notation and process. Two kinds of processes are usually important for software development: the *macro*-process, which concerns management issues; and the *micro*-process, which is concerned with technical details [9]. The micro-process, in part, explains when and how notations are to be used, and how

transformations on specifications can be applied. Micro-processes in formal methods usually include informal steps for how to construct specifications, as well as precise rules for transformation, e.g., for algorithm and data refinement [14,20]. Micro-processes in informal methods include steps and heuristics for constructing specifications, and informal rules for decomposing and transforming specifications into implementations. These processes may need to be supplemented with a macro-process, which includes rules for taking into account business or management requirements.

Two methods may be complementary in terms of their respective processes. This complementarity could manifest itself in several ways. We consider only complementarity in terms of micro-process here (in part, because macro-processes will vary widely from development setting to development setting).

- (1) **Different processes for different tasks.** One method may possess phases of a process that a second method does not. (We might term this *orthogonal* processes.) For example, the formal method predicative programming includes algorithm refinement rules, whereas Larch [10] does not. Larch can certainly be extended with such rules, but in a setting where method integration can be carried out, such extensions are not necessary. Similarly, structured methods like SA/D, and methods like Morgan's refinement calculus [20] are complementary because the former contains processes for diagrammatic modeling and decomposition, while the latter contains rules for refinement.

Methods that are complementary in this manner can often be linked together in sequence; for examples, see [18,25,28]. In this fashion, separate methods are integrated to achieve a new, composite effect.

- (2) **Different processes for the same task.** Two methods may contain similar processes that accomplish the same task. (We might term this *parallel* processes.) For example, Morgan's refinement calculus [20] and predicative programming have processes for algorithm refinement. However, the former process is very different from the latter. In the former, refinement is done in terms of weakest preconditions; in the latter, refinement is done in terms of boolean implication. In many cases, refinement in terms of boolean implication is simpler than in terms of *wp* [14]. A second difference arises in terms of the form of refinement rules. In the refinement calculus, the refinement rules for introducing loops require an invariant and bound function. In predicative programming, a different approach is used: recursive programs rather than looping programs are developed, and invariants are not used explicitly. So even though refinement rules for introducing looping or recursive programs exist in both methods, the approaches inherent in the processes are very different, and very different refinement strategies are used in order to apply them successfully. The methods can therefore be considered as complementary.

In a similar vein, one process for a specific task may be *easier to use* than a second process for the same task. For example, algorithm refinement with Z is usually considered difficult to do in comparison with refinement in Morgan's

calculus, or in predicative programming [14,18]. So process simplicity may provide a useful form of complementarity as well, especially if large problems are to be dealt with.

- (3) **Breadth of process coverage.** Two methods may be complementary if one method presents a process that covers more of the software development process than the second method. For example, SA/D can support the process of software development from requirements through to implementation; predicative programming is typically used for specification through design. A common example of such integrations involves combining formal and informal methods, e.g., [28], so as to scale formal methods up to handle larger problems.
- (4) **Familiarity.** This is similar to familiarity in terms of notation discussed in the previous subsection. Two methods might be complementary if there are two distinct subsets of developers familiar with two distinct processes. This type of complementarity might arise, for example, when introducing formal methods into a development setting where informal methods have been used previously. For example, if the existing process uses UML [8], and B is to be introduced, it may be that the UML users will not immediately be familiar with B's process. So two methods can be complementary in terms of the developers' familiarity with the processes of the method.

If the notations of methods are not complementary, then methods may still be complementary in terms of process. As a further example, in the particular case of propagating formal methods into settings where previously only informal methods were used, it is complementarity in terms of process that can be used to justify integration (e.g., to obtain use of proof techniques). However, if methods are not complementary in terms of notation or process, they may be complementary in other context-dependent ways. We discuss two examples in the next subsection.

### 2.3 *Further kinds of complementarity*

We have discussed two kinds of method complementarity: in terms of notation, and in terms of process. In practice, and in experiments, we may find that there may be several other types of complementarity that do not directly fit under the previous categories.

- (1) **Existence or coverage of tools.** Two methods might be complementary because one method possesses tool support for a specific task, while a second does not. For example, Z possesses support of tools, e.g., for proof or syntax and semantic checking, while predicative programming does not. A specific example of such a complementarity used in practice is in [21], where Larch and predicative programming were combined: Larch tools, e.g., LP, were used behind-the-scenes to prove assertions about an abstract data type that was used

in the predicative specifications. Such an integration was carried out so as to acquire use of tools without having to build them.

Two methods may also be complementary in terms of the coverage of their supporting tools. For example, tools for the Z ‘established strategy’ support proof, syntax and semantic checking, and refinement. Tools for the BON method [30] support diagramming, reverse engineering, code generation, and syntax and semantic checking. We might choose to integrate Z with BON to obtain use of BON tools with Z specifications and vice versa. Z and BON are integrated (via a sequencing of processes; see point (1) in Section 2.2) in [24].

- (2) **Designer requirements.** A designer may have non-functional system requirements (e.g., cost, capability, time) that make methods complementary. Consider a development where designers have different training in widely different methods. One subset of designers may be very familiar with one method, e.g., Booch [5], while a second subset may be very familiar with a different method, e.g., Object Modelling Technique [26]. If both sets of developers are to participate, and limited time is available to train, then supporting both methods, at least at the commencement of the development, may be essential. The two methods might therefore be considered complementary in such a context. Though it should be mentioned that supporting two similar methods in such a circumstance can lead to complications; see [3] for discussion.

## 2.4 Discussion

The previous descriptions of kinds of method complementarity were not intended to be exhaustive. Clearly, there are many other context-dependent ways in which methods could be considered complementary. Indeed, it is only when we are able to study the context in which the integrated methods are to be used that we can justify many useful instances of complementarity.

One inference from our discussions on method complementarity is that better education for software engineers is necessary. This education should ideally include instruction in techniques for determining *which* methods or tools to apply in *which* situations. Instruction or experience may best be acquired through experimentation in applying different methods and tools, and would be abetted with descriptions of applicability or utility of individual methods and tools. Software engineers should not just be able to apply diverse methods and tools: they should also be *methodologists*—as suggested by Jackson [17]—with knowledge and insight on when and when not to apply specific methods and tools.

We also suggest that any general technique for method integration must not place any restrictions on the kinds of methods that are to be combined; complementarity should not be a requirement to use a general method integration technique. The technique should provide guidance on determining whether or not methods to be

combined are indeed complementary, but it should be left to the method integrator (or methodologist) and the intended users of the general method to finally determine whether it is useful to combine methods. It is often the case that only when attempting to combine specific methods does complementarity become clear.

### 3 Conclusions

The issue of *when* to integrate methods may be as important as the issue of *how* to integrate methods. The requirement for methods to be complementary in some way before they should be integrated is vital for justifying specific method integrations, both in terms of the cost of method integration, and in terms of the utility of the integrated methods.

As we have suggested in this paper, methods can be deemed complementary in many ways: in terms of notation, in terms of process, in terms of the existence or breadth of tool support, or in terms of context-dependent criteria such as user preference or developer requirements. This broad spectrum of complementarity suggests that in developing techniques for method integration, we should not fix specific complementarity requirements into the method. Instead, we need to provide support for determining complementarity *during* integration. And even if it is not clear whether specific methods are indeed complementary in some way, we may still find it useful to provide techniques that can be used to combine them, if only to help to validate the method integration process, or to provide support for different kinds of complementarity that we cannot yet determine.

This paper has suggested an informal framework for classifying methods in terms of their complementarity. The next step to take should be to further formalize and clarify this framework, in part by providing clear examples of complementary methods under each of the categories described previously. Work on semantic expressiveness, e.g., [13,21] can be applied for this purpose, as can efforts on theory unification [16], and classification of formalizations of semiformalisms, such as with Structured Analysis [2]. Very useful, in this respect, will be Jackson's notion of *problem frame* [17]. Jackson states that a method is associated with a problem frame; if the problem to be solved fits the frame, the method can be used. For any method, it is vital to know its problem frame, not only to determine whether or not the method will be useful for solving a problem, but also to help determine how the method is complementary with other methods.

**Acknowledgements.** Thanks to Rick Hehner, Pamela Zave, Patrick Place, Tim Denvir, Jonathan Ostroff, and Phil Brooke for helpful comments. Special thanks to the referees for their suggestions. This work was partially supported by the National Sciences and Engineering Research Council.

## References

- [1] Abrial, J.-R., *The B Book* (Cambridge, 1996).
- [2] Baresi, L. and Pezze, M., Toward Formalizing Structured Analysis, *ACM Trans. Software Engineering and Methodology* 7(1), January 1998.
- [3] Berg, W., Cline, M., and Girou, M., Lessons Learned from the OS/400 OO Project, *Comm. ACM* 38(10), October 1995.
- [4] Bicaregui, J., Dick, J., and Woods, E., Quantitative Analysis of an Application of Formal Methods, in: *Proc. Formal Methods Europe '96*, LNCS 1051, Springer-Verlag, 1996.
- [5] Booch, G., *Object-Oriented Design* (Benjamin-Cummings, 1994).
- [6] Bowen, J. and Hinchey, M., Ten Commandments of Formal Methods, *IEEE Computer* 28(4), April 1995.
- [7] DeMarco, T., *Structured Analysis and System Specification* (Yourdon Press, 1979).
- [8] Fowler, M. and Scott, K., *UML Distilled* (Addison-Wesley, 1997).
- [9] Graham, I., Henderson-Sellers, B., and Younessi, H., *The OPEN Process Specification* (Addison-Wesley, 1997).
- [10] Guttag, J.V. and Horning, J.J., *Larch: Languages and Tools for Formal Specification* (Springer-Verlag, 1993).
- [11] Hall, A., Using Formal Methods to Develop an ATC Information System, *IEEE Software*, March 1996.
- [12] Hammond, J., Producing Z Specifications from Object-Oriented Analysis, in: *Proc. Eighth Z User Meeting*, Cambridge, Springer-Verlag, 1994.
- [13] Hehner, E.C.R. and Malton, A.J., Termination Conventions and Comparative Semantics, *Acta Informatica*, 25 (1988).
- [14] Hehner, E.C.R., *A Practical Theory of Programming* (Springer-Verlag, 1993).
- [15] Hoare, C.A.R., *Communicating Sequential Processes* (Prentice-Hall, 1985).
- [16] Hoare, C.A.R. and He, J., *Unifying Theories of Programming* (Prentice-Hall, 1998).
- [17] Jackson, M.A., *Software Requirements and Specifications* (Addison-Wesley, 1995).
- [18] King, S., Z and the refinement calculus, in: *Proc. VDM '90*, LNCS 428, Springer-Verlag, 1990.
- [19] Kronlöf, K. (ed.), *Method Integration: Concepts and Case Studies* (Wiley, 1993).

- [20] Morgan, C.C., *Programming from Specifications*, Second Edition (Prentice-Hall, 1994).
- [21] Paige, R.F., Formal Method Integration via Heterogeneous Notations, PhD dissertation (University of Toronto, November 1997).
- [22] Paige, R.F., A Meta-Method for Formal Method Integration, in: *Proc. Formal Methods Europe '97*, LNCS 1313, Springer-Verlag, September 1997.
- [23] Paige, R.F., Comparing Extended Z with a Heterogeneous Notation for Reasoning about Time and Space, in: *Proc. Z User Meeting '98*, LNCS 1493, Springer-Verlag, September 1998.
- [24] Paige, R.F. and Ostroff, J.S., From Z to BON/Eiffel, in: *Proc. 13th IEEE International Conference on Automated Software Engineering*, IEEE Press, October 1998.
- [25] Polack, F., Whiston, M., and Mander, K.C., The SAZ Project: Integrating SSADM and Z, in: *Proc. Formal Methods Europe '93*, LNCS 670, Springer-Verlag, 1993.
- [26] Quatrani, T. and Chonoles, M., *Succeeding with the Booch and OMT Methods*, (Addison-Wesley, 1996).
- [27] Saeki, M., A Meta-model for Method Integration, *Information and Software Technology* 39 (1998) 925-932.
- [28] Semmens, L.T., France, R.B., and Docker, T.W., Integrated Structured Analysis and Formal Specification Techniques, *The Computer Journal* 35(6), June 1992.
- [29] Spivey, J.M., *The Z Notation: A Reference Manual* (Prentice-Hall, 1992).
- [30] Walden, K., and Nerson, J.-M., *Seamless Object-Oriented Software Architecture* (Prentice-Hall, 1995).
- [31] *Webster's Revised, Unabridged Dictionary*, 1913.
- [32] Zave, P. and Jackson, M., Where do operations come from? An approach to multiparadigm specification, *IEEE Trans. Software Engineering*, 12(7), July 1996.