Lowering the barriers to the use of formal techniques

Dean Allemang

Organon Motives, Inc.
1 Williston Road Suite 4
Belmont, MA 02178
Tel: (617) 484 3383 x13
Fax: (617) 484-3363
Email: dta@organon.com

Abstract

Despite the wide variety of approaches to formal methods for program specification, verification and construction that are available, most programmers still feel that they are not worth the effort. A common reply to this objection is that while it might not be worthwhile to work out a formal description of a single program, that software engineered for reuse can amortize the cost of formalization over the lifetime of a reused component. Unfortunately, widespread institutionalized reuse of software components is not a common practice, which means that opportunities for such amortization are rare. This paper suggests an alternative means for making the use of formal methods tractable for programming, involving the use of a Functional Representation of a program, allowing a more flexible interpretation of what it means to compose devices, thereby relaxing some of the constraints on formal description methods.

Keywords: Functional Representation, Formal specification of components, aspect oriented programming

Workshop Goals: Learn about leading edge work in formal methods for software composition, contribute to a workable vision of how formal methods can impact software construction, herausfinden ob ich immer noch Schwyzertüütsch verstehen kann.

1 Background

Dean Allemang comes from a background in Artificial Intelligence, where he pioneered the application of model-based reasoning, in particular functional reasoning, to software understanding. I quickly found that the formal properties of software were relevant to this study, and in my dissertation work, I integrated a formal approach to software specification with the model-based approach to software understanding.

More reently, Dean Allemang has been involved in research on Domain Modeling for software reuse, and is a co-author of the ODM guidebook [fARSS96]. My latest work has been to integrate a formal functional understanding of software into a design-rationale capture system in the context of the DARPA EDCS programme.

2 Position

2.1 Problem

In this paper, I will address the problem of making formal software composition techniques usable in contexts other than that of a mature component-based software industry. This proposal should not be construed as competing with the development of such an industry, rather as a complementary approach to making formal methods available, thus paving its way. The proposal will make some of the advantages of formal methods more widely available, even in the absence of such an industry.

2.2 Significance of the problem to component-based software engineering

A great deal of work on development of formal methods for software engineering has been done, resulting in many powerful systems for specifying, verifying, and otherwise supporting the development of software. Nevertheless, formal methods have the reputation for being more trouble than they are worth. Programmers are not interested in learning the details of formalisms (aside from their programming languages, of course!), and wonder why they should bother, since the formalization of a problem is often as time consuming as writing a program for it. Proponents of formal methods have a number of replies to this objection, including observations about how the formalization process makes the programming process easier, or reduces maintenance time, communicates program intentions better, or enhances the reusability of software. Despite all these stated (and often demonstrated!) advantages, formal methods remain unpopular.

Another tack taken by proponents of formal methods is to admit that there is an inherent cost in producing formal descriptions, which may well be prohibitive in the context of programming a single system, but that designers of reusable software have a different dynamic, whereby the cost of formalizing a software component can be amortized over multiple use of the component. The formal description of the component can be used to index the component in a library (distinguishing it from other similar components), as well as providing confidence for the reuser that the component will in fact behave as expected, and that the expectations have been expressed and understood clearly.

In order for this argument to work, however, we must suppose that it is possible to construct highly reusable software components, and that it is possible to motivate their use in enough contexts to make this economy of scale work. A number of approaches (e.g., [WOS94]) to making this dream a reality can be found in the literature. This paper takes no position on the eventual possibility of such an industry; however, until such time as such an industry is realized, we will need a story about how formal methods can be used in its absence, if we are to reap the benefits of the work to date on formal methods.

2.3 What theoretical or practical insights might help to solve this problem?

The major insight into resolving this problem is, not surprisingly, that we need to have a more flexible idea of what it means for something to be a *component*, and how they relate to composed groups of components, or *compositions*. In a component-based software industry, we usually think of components as modular pieces of software, which can be composed using some primitive of the programming environment; typically, a mechanism like procedure calls (including remote calls) or method invocation is used to compose components. In object oriented systems, composition is already a bit more flexible; a class definition is often brought into a system by specialization or instantiation. The plan of a component-based industry is that each of these components can be described with some formal statement, and these statements can be used to reason about properties of the composition.

To understand the extension to this notion of component that I propose here, first notice that the notion of component in the last paragraph is described *synthetically*, that is, in terms of how the overall device can be constructed (synthesized) given the components. The proposed notion of composition, in contrast, can be described *analytically*, that is, given a system, what are the parts into which it can be decomposed (analyzed). To avoid confusion, we no longer call these *components*, but instead, *devices*. A device is used for some particular purpose in the system in which it plays a part. This relationship can, of course, be used recursively, i.e., we can talk about devices made up of sub-devices made up of further sub-devices, etc.

One difference between viewing a system in terms of a device decomposition as opposed to viewing it as a composition of components (i.e., analytically rather than synthetically) is that the same system could be decomposed in many different ways; for example, we might view a system as being a combination of a memory manager and a memory mutator (this parlance is common in technical discussions of garbage collectors, where the rest of the program is simply viewed as a "mutator" of memory), or view the same system as a combination of a scheduler and an actuator, or as a combination of a traversal strategy and an action. All of these decompositions refer to the same system, but each brings a different perspective to how it can be understood in terms of constituent devices.

While a synthetic view of describing systems is useful for guiding the construction of a system (e.g., configuring a system out of components, either from a reuse library, or from well-specified custom components), an analytic view is useful for allowing a rich and varied set of descriptions of how software works. In particular, such an approach is useful for program understanding and capture of design rationale. This style of modeling comes from a movement in model-based reasoning in Artificial Intelligence called *Functional Representation*. We are applying a particular functional representation system, called *ZD* [Liv96] to the problem of representing design rationale for software through support of the DARPA Evolutionary Design of Complex Software (EDCS) programme. In the remainder of this section, we will describe how ZD works, and its relation to formal methods.

ZD is a formal system for describing complex intentional artifacts, that is, devices that were designed to achieve a particular purpose. A description in ZD describes a complex device in terms of subdevices, and describes each subdevice in terms of the role it plays in the intended functioning of a higher-level component. Although functional representations in general, and ZD in particular, are based on a new style of modeling the relationship of software systems to their parts, the formal mechanisms that are applicable to component systems are also applicable in a functional representation. ZD is also the name of a program implemented in CLOS, which provides assistance in creating, maintaining, and processing representations of software in ZD. The ZD program is linked to a theorem prover [BM88] to provide access to automatic means for processing formal descriptions of programs.

One benefit of the functional style of modeling software is that each functional decomposition reflects a particular perspective on a system. This means that the formal description of the subdevices, and their interactions, need not specify every aspect of the software. Typically, a small number of concerns is represented with a single decomposition, and only the properties of the software of interest with respect to that concern need be represented.

2.4 Example: The Year 2000 Concern

The following example shows how formal verification methods can be used to solve a very simple version of the Year 2000 concern. The example is intentionally quite simple, but can be modified to show off many of the formal constraint propagation capabilities of ZD.

One of the advantages of a functional representation is that it is possible to describe a system before its design is complete, and still reap gains from the formalism proportionate to the amount of information available for the design. In this example, we pick up the design scenario part way into the design process. The system has two classes of users; employees, who have capabilities on their own workstations for tracking time, and a single Time Account Manager, who collects time reports from all employees and integrates the information into the corporate-level accounting. At this point in the design process, certain decisions have been made:

- part of the system will be responsible for allowing an employee to record the amount of time spent on different jobs. This capability has been adapted from public domain code (source code is available),
- the system will be able to save timecard information to a file in a standard format,
- part of the system will perform time accounting functions. This is based on a commercially available product that provides the necessary accounting services,
- the time accounting software provides a service for reading timecard information from a file in a standard format, and
- these two devices will communicate through file exchange, in one of the formats acceptable to the commercial accounting package.

The details of the file transfer facility have not yet been decided; in particular, we have not decided if the two systems will run in a shared file space, or use a network protocol (say, SMTP or FTP) to exchange files.

We describe this design by identifying three devices; an overall device, called IOP, which coordinates two other devices, which we call TIMEFLIES and TIMECARD. Figures 1 and 2 show an excerpt of the ZD formalism that describes this decomposition. The viewpoint taken by this decomposition is particularly suited to dealing with concerns about the communication between the employee timekeeping and the account manager time calculations. In this example, we will use this to express year 2000 constraints on the various devices. We cannot give a complete description of the formalism in this short paper, but we can describe how the particulars of this example appear in the formal ZD description.

ZD represents devices in terms of the *Services* they provide. A Service is specified as a number of *functions*. In ZD, functions are specified in terms of preconditions and postconditions, which are indicated in the formalism with the keywords **If** and **ToMake** respectively. The clauses after the keywords **If** and **ToMake** are expressed in the language of the Boyer-Moore theorem prover NQTHM [BM88]. In the figures, we can see Service specifications (**ServiceSpec**) for three devices, IOP, TIMEFLIES and TIMECARD. There is no Service specified for the file transfer medium, since that has not been designed yet.

Much of the information about types and unification has been left out of figures 1 and 2. Of the remaining formalism, the greatest interest for this example is the annotated finite state diagram annoted by **(1)** in the figure. This shows how the IOP device coordinates devices in three roles, namely the roles RECORDING-SYSTEM, MEDIUM and ACCOUNTING-SYSTEM. The diagram shows that in order for the IOP to work as specified, it will have to call upon the EXPORT function of the RECORDING-SYSTEM, the TRANSFER function of the MEDIUM, and the IMPORT function of the ACCOUNTING-SYSTEM. Finally, from the line labeled **(2)** onwards, we see that the TIMECARD device will fill the RECORDING-SYSTEM role, and that the TIMEFLIES device will fill the ACCOUNTING-SYSTEM role.

Given this functional decomposition, ZD provides a number of mechanisms for describing constraints among the various devices. We will describe only one of them here.

In each of the four lines labeled **(3)**, we see a clause introduced by the keyword **Provided**. This indicates a precondition of the function that is to be interpreted dynamically, that is, the function annotated with a **Provided** clause will provide the specified functionality, provided that the condition holds in the context when the functionality is required. In this case, the proviso gives information about the year 2000 assumptions of each of the constituent devices.

ZD's proviso propagation mechanism maps these conditions to the corresponding roles in the state diagram shown by **(1)**. It then propagates them backwards, making any necessary substitutions of variables required by the state changes expressed in the finite state diagram. In this simple example, the proviso year-precision($data$) $<$ `100` of the function IMPORT of the role ACCOUNTING-SYSTEM is propagated backwards; the IMPORT function will be achieved as specified just whenever the data has the appropriate property. The function EXPORT of the role RECORDING-SYSTEM also places a constraint on the year-precision($data$); namely, year-precision($data$) $<$ `10000`. Since this condition is strictly stronger than the IMPORT condition, ZD continues propagating with only the stronger condition. When ZD reaches the front of the finite state diagram in (1), it is in a position to state that this explanation for how the roles will be coordinated will work as specified, just in case year-precision($data$) $<$ `100`. Since this state diagram is the explanation for the SUBMIT function of the IOP service, ZD places this condition as a proviso on that function. If the IOP were to play a role in a larger device, this proviso would be propagated further.

2.5 Applications and Limitations

The simple example of ZD shown above illustrates one of the mechanisms for maintaining the consistency of a functional representation. We have applied this formalism in the context of design rationale capture in the following way: during a top-down design process, a designer proposes a particular decomposition of a system. The details for how each of the parts of this system (represented in ZD as devices) is to be realized has not yet been determined. However, it is possible for the designer to specify some constraints about the fillers of these roles at this time. As other designers on the team work to fill in the particular roles, ZD can automatically verify whether a particular combination of choices is satisfactory, according to the original designer's expressed constraints.

We have demonstrated this apporach on a handful of examples, which have shown that it is applicable to simple configuration problems such as the Year 2000 example shown here, as well as more involved analyses of iterative behavior at both the implementation and architectural description levels. The real limitation in the method seems to be the expressiveness of the finite-state language used to describe the relationships between functional components; our experiments have shown that this limitation is more restrictive than the choice of NQTHM as a theorem prover. The architecture of the ZD implementation was designed so that the theorem prover could be easily exchanged for another; however, we have not found a need to do so. The information available from the functional representation has proven so far to be sufficient to provide the theorem prover enough information to avoid any impasses in its proofs.

3 Comparison

Functional representation represents quite a departure in modeling philosophy compared to other model-based reasoning paradigms in AI. In terms of software composition, its application also represents something of a departure in terms of viewing program composition analytically rather than synthetically, as described above. However, an analytic approach to software engineering is not new; there are a number of movements that share this point of view.

There has been a recent movement in software engineering towards explicit manipulation of software architectures, and the introduction of software architecture definition languages (ADLs) (e.g., Rapide [LKA$^+$95] and ACME [GMW96]). An architecture definition differs from a program specification in that it does not describe the particular components from which a system will be built; instead, it defines the types of components and the responsibilities of the connections between them. This feature of an ADL has cause many of them to be described informally as "box-and-arrow languages." Like a functional representation, an architecture can be defined and evaluated before any decision has been made to realize any of it in terms of actual software components.

In this respect, ZD plays a similar role in the design process as an architecture definition language (another functional representation language, SBF [Str94] is being used explicitly as an ADL). We have also done some preliminary research in which ZD has been used to represent a reference architecture for two-phase commit (a problem that is often used to show off ADL features). ZD's flexible treatment of constraints seem to make it possible to represent certain aspects of this architecture more flexibly than is possible in other ADLs.

A similar approach to system design can be found in the expert systems development community in frameworks for formal knowledge system design. KARL [Fen95], $(\mathrm{ML})^2$ [vHB92] and Desire [Tre94] are frameworks for working "at the knowledge level", which entails specifying a expert system without reference to implementation details. These frameworks are typically based on logical inference languages. While they are very flexible in their ability to describe complex interactions, and have formal support (often supported by inferencing tools), they are typically not component oriented, and do not provide any insight into how formal methods can be applied to component-based software engineering.

All of these analytical approaches, including functional representations, share a problem that is particularly important to component-based software, namely, although it is possible to evaluate the correctness of an architecture specified in an ADL, or a knowledge base specified in a language like KARL, or a functional representation of a system, this leaves open the problem of deciding whether a particular piece of software is accurately described by this formal description. In the case of the knowledge based design formal systems, there is a specific step in the lifecycle, called the "design phase," where this gap is crossed. In a component-based software reuse industry, this bridge need not be crossed, since the formal descriptions correspond directly to pieces of code. Program generation approaches (e.g., Jakarta [Bat97]) address this problem directly, by providing a system whereby a design can be developed using a high-level specification language, from which software can be automatically generated.

A particularly interesting form of generative technique goes under the name "adaptive programming" or "aspect oriented programming" [Lie96, Kic97]. In an adaptive program, the system is decomposed not into modules that inter-operate according to some calling sequence provided by the programming language, but rather into interacting *aspects* or *concerns*. For example, in Demeter, an object-oriented program is divided into a traversal strategy (a "strategic aspect") and a set of actions ("visitors"). These two aspects interact in complex but predictable ways. The Demeter system generates a program from independent descriptions of these two aspects.

In the sense described above, the multiple-aspect decomposition used by Demeter is a functional decomposition. Although Demeter gets much of its power from the independence of these aspects, in reality, the correctness of a traversal strategy depends to some degree on the details of how the traversed nodes are to be visited. Although Demeter can synthesize a program from these aspect descriptions, it cannot reason about the correctness of the composition at the higher level. We are currently collaborating with the Demeter team to apply ZD to this kind of composition.

4 Biography

Dean Allemang completed his PhD at The Ohio State University in 1990 applying functional representation to software. Between 1990 and 1995, he worked at the University of Zurich, the

Istituto Dalle Molle di Studi sull'Intelligencia Artificiale in Lugano, and the Swiss Federal Institute of Technology in Lausanne on knowledge acquisition and case-based reasoning for expert systems. He completed his tour of Switzerland with a year and a half at the Research and Development Division of Swiss Telecom, where, among other projects, he worked with Beat Liver on functional representation of software. In 1996, he joined Organon Motives where he contributed a chapter on knowledge acquisition to the ODM guidebook. His recent work has focused on the use of functional representations of software in the capture of design rationale information for software evolution.

References

[Bat97] Don Batory. Intelligent components and software generators. Technical Report 97-06, University of Texas at Austin, 1997.

[BM88] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook.* Academic Press, New York, 1988.

[fARSS96] Software Technology for Adaptable Reliable Systems (STARS). Organization domain modeling (odm) guidebook, version 2.0. Technical Report STARS-VC-A023/011/00, STARS, 1996.

[Fen95] Dieter Fensel. *The Knowledge Acquisition and Representation Language KARL.* Kluwer Academic Press, Boston, 1995.

[GMW96] David Garlan, Robert Monroe, and David Wile. Acme: An architecture interchange language. *IEEE Transactions on Software Engineering*, 1996.

[Kic97] Gregor Kiczales. Aspect-oriented programming. In *Proceedings of WISR 8*, Columbus, OH, 1997.

[Lie96] Karl Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, 1996.

[Liv96] Beat Liver. *A functional model of software and its application in troubleshooting, design and reuse.* PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 1996.

[LKA+95] David Luckham, John Kenney, Larry Augustin, James Vera, Dough Bryan, and Walter Mann. Specification and analysis of system architecture useing rapide. *IEEE Transactions on Software ENgineering*, 21(4):336–355, 1995.

[Str94] Eleni Stroulia. *Failure-Driven Learning as Model-Based Self-Redesign.* PhD thesis, Georgia Institute of Technology, 1994.

[Tre94] J. Treur. Temporal semantics of meta-level architectures for dynamic control of reasoning. In Fribourg et al., editor, *Logic Program Synthesis and Transformation - Meta Programming in Logic.* Springer Verlag, 1994.

[vHB92] Frank van Harmelen and J. Balder. $(ML)^2$: A formal language for kads conceptual models. *Knowledge Acquisition*, 4(1), 1992.

[WOS94] Bruce Weide, William Ogden, and Murali Sitaraman. Recasting algorithms to encourage reuse. *IEEE Software*, pages 80–88, September 1994.

ServiceSpec: IOP
Service: SUBMIT (funimp: SUBMIT)

If: loaded(*ts-e*, *etc*)
ToMake: loaded(*ts-tam*, *etc*)

DevImp: SUBMISSION
Funimp: SUBMIT (devimp: SUBMIT)

$$
\begin{array}{cl}
\text{loaded}(\textsc{ts-e}, \textsc{etc}) & \\
\downarrow & \text{RECORDING-SYSTEM.EXPORT} \\
\text{in-form}(\textsc{format}, \textsc{etc}) \wedge \text{present}(\textsc{m-e}, \textsc{etc}) & \\
\downarrow & \text{MEDIUM.TRANSFER} \\
\text{in-form}(\textsc{format}, \textsc{etc}) \wedge \text{present}(\textsc{m-tam}, \textsc{etc}) & \\
\downarrow & \text{ACCOUNTING-SYSTEM.IMPORT} \\
\text{loaded}(\textsc{ts-tam}, \textsc{etc}) &
\end{array} \tag{1}
$$

ReqSpec: MEDIUM
Requirement: TRANSFER
If: present(*source*, *message*)
ToMake: present(*destination*, *message*)

ReqSpec: ACCOUNTING-SYSTEM
Requirement: IMPORT
If: in-form(*format*, *etc*) $\wedge$ present(*m-tam*, *etc*)
ToMake: loaded(*ts-tam*, *etc*)
ReqSpec: RECORDING-SYSTEM

Requirement: EXPORT
If: loaded(*ts-e*, *etc*)
ToMake: in-form(*format*, *etc*) $\wedge$ present(*m-e*, *etc*)

Figure 1: Time accounting example in the ZD formalism.

Glue **(2)**
top: RECORDING-SYSTEM **bottom:** TIMECARD
Representation:

$$\rho(x) = \begin{cases} LOADED & \text{if } x = LOADED \\ TS - E & \text{if } x = TIMECARD \\ IN - FORM & \text{if } x = IN - FORM \\ FORMAT & \text{if } x = COMMA \\ ETC & \text{if } x = DATA \end{cases}$$

Function bindings EXPORT – EXPORT-COMMA

ServiceSpec: TIMECARD

Service: EXPORT-TAB
If: loaded($timecard$, $data$)
ToMake: in-form(tab, $data$)
Provided: year-precision($data$) $<$ `10000` **(3)**

Service: EXPORT-COMMA
If: loaded($timecard$, $data$)
ToMake: in-form($comma$, $data$)
Provided: year-precision($data$) $<$ `10000` **(3)**

Glue
top: ACCOUNTING-SYSTEM **bottom:** TIMEFLIES
Representation:

$$\rho(x) = \begin{cases} LOADED & \text{if } x = LOADED \\ TS - TAM & \text{if } x = TIMEFLIES \\ IN - FORM & \text{if } x = IN - FORM \\ FORMAT & \text{if } x = COMMA \\ ETC & \text{if } x = DATA \end{cases}$$

Function bindings IMPORT – IMPORT-COMMA

ServiceSpec: TIMEFLIES

Service: IMPORT-TAB
If: in-form(tab, $data$)
ToMake: loaded($timeflies$, $data$)
Provided: year-precision($data$) $<$ `100` **(3)**

Service: IMPORT-COMMA
If: in-form($comma$, $data$)
ToMake: loaded($timeflies$, $data$)
Provided: year-precision($data$) $<$ `100` **(3)**

Figure 2: Time accounting example in the ZD formalism.