The two dimensions of an architecture

Tommi Mikkonen
Software Systems Laboratory
Tampere University of Technology
P. O. Box 553, FIN-33101 Tampere, Finland
Tel. + 358 3 365 3815, Fax + 358 3 365 2913
Email tjm@cs.tut.fi

Abstract

A two-dimensional view on software architectures is introduced. One of the dimensions is the implementation-oriented view based on component structure, and the other is defined by abstractions and logic of collaboration. We argue that unless the view based on collaboration is rigorously taken into account in early phases of the development, architectures will be based on informal expectations and implementation bias. This bias may then blur the logic of the specification, whereas an adequate use of abstractions of collaboration as basis for architecture will eliminate superfluous difficulties in design and verification.

Keywords

Logical layers, formal methods

1 INTRODUCTION

A current trend in computing, the shift from sequential programs to interaction and distributed cooperation, is tremendously increasing the complexity of software systems (Wegner 1997). The ability to deal with collaboration of multiple parties, however, has not increased accordingly. For instance, while the flavors of modularity suited for implementing conventional systems have been elegantly discussed in (Parnas 1972), modularity applicable to addressing abstract cooperation of potentially distributed parties has received virtually no attention until recently. Currently, the increasing use of design patterns (Gamma *et al.* 1995) can be regarded as an attempt to address such issues.

Conventional components, like processes or objects, and collaborative aspects embedded in them can be used to define the two dimensions of an architecture, addressing the structure and the logic of the system, respectively. This paper introduces a way to use the logical dimension to derive architectures in

a rigorous manner. The origins of the approach are in joint actions (Back *et al.* 1989), and in the DisCo method (Järvinen *et al.* 1990), which is based on the Temporal Logic of Actions (Lamport 1994). In this paper, the discussion is at the level of fundamental ideas and the underlying philosophy only. In particular, all language issues have been omitted.

The rest of this paper is structured as follows. Section 2 discusses the shift of concern from components to their cooperation. Section 3 forms the core of this paper by defining a way to compose specifications with abstractions of collaboration as units of modularity. Finally, Section 4 concludes the paper.

2 FROM COMPONENTS TO COOPERATION

This section discusses the two dimensions of an architecture. The discussion starts with conventional architectural definitions, which are typically given in terms of programming-level components constituting the system. Then, the emphasis shifts from concepts local to individual components to multiparty cooperation.

2.1 Elements of a structural architecture

Software systems are artifacts that consist of program units. These units are often viewed as *open systems* in the sense that they offer services to their environments, and encapsulate some attributes. While this approach has been adapted to support concurrency, it is essentially based on a sequential model of execution. Each service can be seen as a procedure that produces a certain output for a given input, possibly taking into account the current state of the associated program unit. The available services then form the unit's behavior. With such elements, the task of an architect is to define a satisfying way to connect the program units with data and control flows.

Such an approach emphasizes the role of interfaces connecting the system and its environment, as well as the interfaces enabling interparty communication. After identifying convenient interfaces, the focus is set on internals of individual architectural units, like processes or objects. These units and the abstractions embedded in them can then be designed and implemented in isolation. Furthermore, when integrated, their composition should satisfy the original requirements.

With this view, there are usually no abstractions that would dependably extend beyond individual program units. Moreover, the behavior of a complete system is available only in an informal sense until the units have been formally specified (or implemented) and put together. Therefore, there is no rigorous definition of what to implement until the implementation is completed. This results in inability to explicitly formalize expectations related to collaboration in an abstract fashion, despite acknowledging that most of

today's problems call for something that extends across objects and ties them together (Coplien 1997). Moreover, as the environment need not be implemented, the assumptions related to it may never be formalized.

While novel approaches to architectures, like (Allen *et al.* 1997), enable rigorous reasoning on cooperation of components, an architecture formed with components is still required until collaboration can be dependably addressed. Sophisticated approaches, like Corba (Vinoski 1997) and the Time-Triggered Architecture (Kopetz 1997), place the focus on an implementation structure provided by the standard architecture, but fail to support abstract specification of the logic of the application. As reasoning about collaboration is hard with implementation-level concepts (Kurki-Suonio *et al.* 1998a), properties extending across objects should be taken into account rigorously at an abstract level.

2.2 Towards an architecture based on collaboration

Decomposition into implementable modules is not essential for rigorous reasoning (Lamport 1997). By shifting the focus from components to their collaboration, a system can be viewed to consist of *logical layers* that introduce temporal properties of a system in a modular fashion. Such properties can be safety or liveness properties, formalizing statements of the form "Something bad will never happen" and "Something good will eventually happen," respectively.

With such an approach, each logical layer can be interpreted as a description of a *closed system* that can be observed but not affected from outside. Unlike open systems, closed systems require no external control or data flows. Instead, such systems are self-contained, i.e., they are capable of performing operations, and their environment assumptions are explicitly included in the associated specification in an operational fashion. This interpretation enables us to define a layer, possibly involving distributed parties, and observe its properties in isolation. Composition of such layers then provides a way to define how the layers interfere, resulting in an architecture based on collaboration rather than static component structure.

Related approaches include program slicing (Weiser 1982) and projections used in program verification (Lam *et al.* 1984). Unlike in these approaches, however, we compose new specifications with layers, not decompose existing systems with them. A somewhat restricted form of similar layering can also be seen in the constraint-oriented specification style, proposed in connection with LOTOS (Bolognesi *et al.* 1987).

3 DERIVING AN ARCHITECTURE WITH LAYERS

This section discusses the derivation of an architecture with logical layers. The emphasis is on the mechanisms and abstractions that support layer-based specification of potentially distributed aspects. Towards the end of the section, the emphasis shifts to necessary implementation decisions.

3.1 Abstractions of behavior

Software specification requires abstractions. In order to support an abstract view on temporal behaviors, we introduce the notion of an *action* as a primitive operation whose execution is atomic. Thus, an action is a "step" relation between two sets of values of state variables. When an action can be executed, it is *enabled*.

Individual state variables are gathered to objects. Collaboration of the objects is then defined by giving actions that affect variables in them. Actions are defined in a manner where objects can take different roles in executions. During an execution of an action, objects taking the associated roles can be accessed freely, disregarding issues related to direct implementability.

In the execution model, actions are executed in an interleaving fashion. The action to be executed next is nondeterministically chosen from those that are enabled for some combination of participating objects. Similarly, if there are several sets of objects for which an action can be executed, one of them is selected in a nondeterministic fashion. This nondeterminism enables postponing of design decisions with respect to control and data flows at abstract level.

With this definition, actions specify safety properties only. In order to incorporate liveness properties in specifications, fairness requirements are allowed to be given with respect to roles defined for objects in actions. Such requirements ensure that if an action is repeatedly enabled so that an object can take a role with fairness requirements, the action will eventually be executed for the object.

3.2 Composition and refinement

Specifications can be made more precise by refining them. We restrict ourselves to refinements that apply *superposition*. Such refinements can only introduce new variables, not remove existing ones. With superposition, safety properties are preserved by construction (Järvinen *et al.* 1990). Preservation of liveness properties requires additional proofs.

Every refinement adds a new logical layer to the specification. When layers are conjoined, the resulting specification is a refinement of all of them, ensuring the satisfaction of component layers. Therefore, independent issues

can be specified separately, and later incorporated in the same specification without risking (safety) properties. This results in straightforward aspect-oriented specification, where unrelated issues can be separated in the specification phase even if their implementations reside in the same program unit, as demonstrated in (Kurki-Suonio *et al.* 1998).

Although all refinements are layers, their conceptual meanings may vary. For instance, a layer can introduce a new aspect, or define how an existing aspect is implemented with lower-level mechanisms. The uniform view also supports rigorous reasoning, as all proofs can be carried out in a similar fashion.

3.3 Utilizing abstractions

When discussing multi-party cooperation, component-based approaches lead to laborious formalizations, where the focus is set on the implementation-level communication of individual program units (Kurki-Suonio *et al.* 1998a). With logical layers, we have adopted a viewpoint where the development is initiated with high-level conceptual abstractions (Kellomäki *et al.* 1998).

In order to create valid abstractions, they need to be explicitly incorporated in the specification, with an enforcement policy to ensure that they will not be accidentally invalidated. As long as abstractions are solely based on program units, this enforcement is easy and natural to comprehend. However, the same requirement to enforce abstractions remains valid for abstractions of collaboration. Therefore, an effort is needed to ensure the validity of existing abstractions when refining the specification into an implementable form (Kurki-Suonio *et al.* 1998b).

Logical layers support validation of collaboration at an abstract level. Because abstractions need not be directly implementable, rigorous reasoning and early validation of the underlying domain model can be considerably simplified by omitting unnecessary implementation details. Moreover, when refining the specification into an implementable form, there is a rigorous definition of correctness given at an abstract level.

3.4 Refinement towards an implementation

With logical layers, application logic can be separated from implementation details. While design decisions on convenient implementation components still need to be made, a layer-based specification supports the decisions by providing a formal model on the intended behavior of the system. The design can thus be based on a rigorous specification instead of informal expectations and premature implementation bias.

When top-level abstractions of collaboration have been found satisfying,

the effort shifts to the derivation of an implementation of abstract collaboration with the available lower-level concepts of cooperation. During this derivation, the focus is on verification of the validity of abstractions. Due to this verification, we can dependably refine a logical architecture, which defines collaboration at an abstract level, into a form that is implementable with available communication mechanisms (Kurki-Suonio *et al.* 1998b). Conventional approaches use similar ideas for mapping file abstraction to its hardware implementation, for instance. We, however, use layers to define abstract collaboration and its correct implementation.

In practice, when more concrete variables are used to implement their abstract counterparts, it is necessary to give invariants that define the relation of abstract and implementation-level variables. Such invariants turn the abstract variables into non-primitive state functions whose explicit incorporation into an implementation is not necessary. Refinement towards an implementation will result in a situation where the specification more and more anticipates the final structural architecture, with rigorous enforcement that the abstractions will be satisfied.

When the design has progressed to a level where all structural components and their behavior can be identified, an implementation can be made in a conventional manner. Obviously, an ideal case would be an automatic code generator, implementing specifications with some feasible implementation mechanisms. This, however, seems to be unrealistic in the general case, due to potential variety of desired implementation mechanisms.

4 CONCLUSIONS

Abstract collaboration and its implementation with available communication primitives were discussed in the context of logical layers. Such a dimension of an architecture, orthogonal to traditional component-based architectures, supports design and verification by enabling separation of application logic and its structural implementation. The validity of abstractions used to constitute the architecture is enforced by embedding a discipline in the associated specification method, thus facilitating the definition of collaboration of potentially distributed parties at an abstract level.

REFERENCES

Allen, R., Garlan, D. (1997). A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology 6*, 3, July 1997, 213–249.

Back, R.J.R., Kurki-Suonio, R. (1989). Decentralization of process nets with a centralized control. *Distributed Computing 3*, May 1989, 73–87.

Bolognesi, T., Brinksma, E. (1987). Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59.

Coplien, J.O. (1997). Idioms and patterns as architectural literature. *IEEE Software*, January 1997, 36–42.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Järvinen, H.-M., Kurki-Suonio, R., Sakkinen, M., and Systä, K. (1990). Object-oriented specification of reactive systems. In *Proceedings of the 12th International Conference on Software Engineering*, IEEE Computer Society Press, 63–71.

Kellomäki, P. and Mikkonen, T. (1998). Modeling distributed state as an abstract object. Accepted to International IFIP Workshop on Distributed and Parallel Embedded Systems, to be published.

Kopetz, H. (1997). *Real-Time Systems. Design Principles for Distributed Embedded Applications*, Kluwer.

Kurki-Suonio, R., Katara, M. (1998). Real time in a TLA-based theory of reactive systems. In *Proceedings of the First International Symposium on Object-Oriented Real-Time Distributed Computing*, IEEE Computer Society, 186–195.

Kurki-Suonio, R., Mikkonen, T. (1998a). Liberating object-oriented modeling from programming-level abstractions. *Object-Oriented Technology: ECOOP'97 Workshop Reader*, (ed. J. Bosch, S. Mitchell), Springer-Verlag LNCS 1357, 195–199.

Kurki-Suonio, R., Mikkonen, T. (1998b). Abstractions of distributed cooperation, their refinement and implementation. In *Proceedings of International Symposium on Software Engineering for Parallel and Distributed Systems*, (ed. B. Krämer, N. Uchihira, P. Croll, S. Russo), IEEE Computer Society, 94–102.

Lam, S.S., Shankar, A.U. (1984). Protocol verification via projections. *IEEE Transactions on Software Engineering*, SE-10(4), July 1984, 325–342.

Lamport, L. (1994). The temporal logic of actions. *ACM Transactions on Programming Languages and Systems 16*, 3, May 1994, 872–923.

Lamport, L. (1997). Composition: a way to make proofs harder. Digital Research Corporation, Technical Note 1997-030a.

Parnas, D.L. (1972). On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(12), December 1972, 1053–1058.

Vinoski, S. (1997). CORBA: Integrating diverse applications within distributed heterogenous environments. *IEEE Communications*, 14(2), February 1997, 80–91.

Wegner, P. (1997). Why interaction is more powerful than algorithms. *Comm. ACM*, 40(5), May 1997, 80–91.

Weiser, M. (1982). Programmers use slices when debugging. *Comm. ACM*, 25(7), July 1982, 446–452.

5 BIOGRAPHY

Tommi Mikkonen teaches and does research in Software Systems Laboratory at Tampere University of Technology. After receiving M.Sc. (1992) and Lic. Tech. (1995) degrees, he took a brief glimpse on dependable computing by working for space industry, and has recently rejoined the university. Currently, he is finishing his doctoral thesis on abstractions and modularity applicable to the specification of interaction.