MODELING DISTRIBUTED STATE AS AN ABSTRACT OBJECT

Pertti Kellomäki and Tommi Mikkonen
Tampere University of Technology
P.O.Box 553, Fin-33101 Tampere, Finland
{pk, tjm}@cs.tut.fi

1. Introduction

Object-oriented software development emphasizes the use of abstractions. Inheritance is used for factoring common structure and functionality to abstract classes, providing abstract interfaces to objects. This helps in avoiding premature implementation bias.

Current methods of software engineering, such as OMT (Rumbaugh *et al.*, 1991), focus on the behavior of implementation level objects. In distributed systems, however, we can also identify abstractions that arise from the cooperation of several objects. These abstractions help in explaining the collective behavior of objects in more high level terms. An example of such abstractions is the way various components of a telephone network cooperate in order to provide a phone call.

Formal approaches to object-oriented development like Z++ (Lano, 1991) or VDM++ (Durr *et al.*, 1992) also usually focus on the behavior of single objects. In this paper, we use the DisCo specification method (Järvinen, 1992), (Järvinen *et al.*, 1991), which has been explicitly designed for expressing collective behavior in terms of synchronous actions. The DisCo language has a formal basis in the joint action theory (Back *et al.*, 1988), (Järvinen *et al.*, 1990) and Temporal Logic of Actions (Lamport, 1994), which gives DisCo specifications an unambiguous meaning. We can thus reason informally or formally about the behavior described by a DisCo specification. Moreover, a DisCo specification can be used as a precise reference to answer questions about the intended behavior of the system.

F. J. Rammig (ed.), *Distributed and Parallel Embedded Systems*

We see the possibility of early validation and verification of behavioral properties as the main advantage of our approach. When abstractions are expressed using a precise notation, we can use them for more than just conveying informal intuition. Animation can be used when specifications are validated by experts of the application domain, and informal or formal reasoning may be used for verifying temporal properties of the specification.

The rest of this paper is structured as follows. Section 2 introduces the basics of the DisCo method. Section 3 discusses abstractions and their temporal behaviors, and gives an example of such an abstraction. Section 4 discusses validation and verification using explicit abstractions of cooperation. We focus on effects on early stages of development. Related work is presented in Section 5, and Section 6 lists the main conclusions.

2. The DisCo method

In this section we give a short overview of DisCo, a specification method and language for the specification of reactive systems. For the purposes of this paper, the most important issues are the closed world principle, superposition, and preservation of properties in a refinement.

The approach adopted in DisCo is to begin the development by identifying variables and interactions whose role is fundamental to the behavior. The variables at this level may represent implementation-level entities such as phones, or abstractions such as phone calls. Details are then added in a modular manner, allowing implementation of the high-level abstractions by using more concrete objects.

In DisCo, variables are defined in terms of classes that are patterns for objects. Each object can be understood as a structured variable. Objects may contain local variables, references to other objects, and nested statemachines similarly to Statecharts (Harel, 1997). In addition, objects can be associated with each other by using relations.

The underlying computational model is based on *joint actions*, where a number of *objects* may exchange information. An action may be executed whenever suitable objects for which the action *guard* evaluates to true exist. Action execution is atomic, and nondeterministic choice is used for modeling concurrent activities. Actions are given in the format:

```
name(participants):
when guard
-> body,
```

where *participants* is a list of formal names for the participating objects, *guard* is a boolean-valued expression, and *body* is a sequence of statements.

At a high level of abstraction, it is not always clear who initiates operations, especially if several objects are involved. The joint action approach considers this to be an implementation detail that can be deferred, and therefore does not indicate which component initiates the execution of an action. Adopting this simplification liberates us from programming-level abstractions, making the approach suitable for

high level specification where only the information exchanged and the commitment of the communicating parties is of interest (Kurki-Suonio *et al.*, 1997).

2.1. Closed-World Principle

The closed-world principle is the philosophical backbone of the DisCo methodology. Each DisCo specification is a complete description of a system in the sense that it describes all the possible changes to the variables introduced in the specification. The closed world principle implies that every DisCo specification models both a system and its environment, thus facilitating reasoning about their collective behavior.

Any introduction of new properties, referred to as a refinement, reveals more properties of the system, but cannot allow behaviors that the more abstract level does not allow. Thus, refinements can be understood analogous to dimensions of vectors. The analogy extends to the possibility to project the behavior of the system into a certain set of refinement steps.

2.2 Superposition and Preservation of Properties

DisCo specifications may be nondeterministic. Refinement steps may be used to superimpose stronger constraints on nondeterminism, by introducing new variables and operations on them. Operations may only assign to variables introduced in the same refinement step. Thus, a DisCo refinement can be understood as a system-wide layer, that applies advantages of program slices (Weiser, 1992) to specification. The method enforces that *safety properties* ('Something bad will never happen') introduced in one specification layer cannot be invalidated in subsequent layers. As guards of actions can be made stronger, preservation of *liveness properties* ('Something good will eventually happen') are not guaranteed by construction.

With such refinements, a specification can be "broadened" by providing new aspects as well as "deepened" by introducing new details on how some operations are implemented. The former allows incremental specification when the specification is still incomplete, whereas the latter enables implementation of an operation by using a more concrete representation. In practice, there may not always be clear distinction between the two.

3. Cooperation as an object

In a distributed system, the state of a high level abstraction, such as a phone call, may be distributed in the local states of several objects. We make the state of the abstraction explicit by representing the abstraction as an object. This helps in focusing on the collective behavior instead of the details of a distributed implementation. The explicit object describes the state of several distributed objects in a centralized fashion.

The use of a specification language with precise semantics enables us to use the abstractions for more than informal sketching. Behaviors can be expressed and validated with abstractions, with the assurance that the high level properties are not invalidated in later refinement steps.

In the following, we give an example of an abstraction of cooperation, and discuss its distributed implementation. Consider a mobile phone system that consists of phones and base stations. Each phone can be linked to a base station via a radio link, and base stations can establish links among themselves to route calls. We impose the rather unrealistic restriction that only one phone at a time may be connected to a base station.

When a phone moves about, it may disconnect from one base station and connect to another. If the phone is involved in a call, the connections between base stations are modified accordingly. Thus, a handover is possible, where a phone disconnects from one base station and connects to another during a call. A phone call consists of three connections: one connection between the base stations, and connections between the phones and the base stations. Over the duration of a call, any one of these connections may be changed.

3.1 Abstract Level

Even though a phone call is clearly a central notion in explaining how the system works, it is not part of the class hierarchy of the specification of the mobile phone network. It would not make much sense to inherit a phone call class in either phones or base stations. Nor is a phone call a subsystem composed of the connections, since the connections may be changed on the fly. Rather, a phone call is the result of cooperation between the phones and base stations.

By making this cooperation explicit, we can create a simpler model that can be used for verification and validation of high level properties of the system. We next give a specification of the mobile phone system where the call abstraction is represented as an explicit object. In this specification, a phone may be on or off, indicated by boolean variable *on*. In addition, each phone call contains a data field *parties*, a set of phones. The actions of the specification can be given as follows:

```
connect(p1, p2 : phone; c : call):
when c.parties = { }
   and p1.on
   and p2.on
   and forall c2 : call: (p1 in c2.parties or p2 in c2.parties) = false
   -> c.parties := {p1, p2},

disconnect(c : call):
   when not c.parties = { }
   -> c.parties := { },

on(p : phone):
   when not p.on
   -> p.on := true,

off(p : phone):
   when p.on
   -> p.on := false.
```

The model given by the specification is highly nondeterministic. Two phones that are in state *on* and not involved in a phone call may arbitrarily be connected, and a phone call may be disconnected at any time. However, the model can already be used to give exact answers to questions about the behavior of the system. For example, we can ask whether it is possible for more than two phones to be involved in a call, or whether "zombie" calls where one of the phones is in state *off* are possible. Validation and verification of such issues are discussed separately in Section 4.

The abstract phone call serves as a convenient place for storing various attributes a call might have: who is paying for the call, what is the duration of the call, etc.

3.2 Distributed Level

Once we are satisfied that the abstract specification captures our informal requirements, we use DisCo refinements to derive a less abstract specification of the system. For instance, the abstract call discussed earlier can be implemented by three connections: two between phones and base stations, and one between two base stations. We omit the specification here for brevity.

Due to the use of superposition, the specification of the distributed level formally contains the call objects introduced at the abstract level. However, we can justify an implementation where calls do not exist as implementation level objects if we can show how the state of each call can be computed. This also implies that the properties expressed in terms of abstract calls hold in such an implementation. The following invariant shows how the state of a call is computed (*connected* is the obvious relation). This may be established either informally of formally, as discussed in Section 4. The invariant is:

```
forall c in call, p1, p2 in phone:
  c.parties ={p1, p2} implies
    exists b1, b2 in basestation:
          connected(b1, b2)
      and connected(p1, b1)
      and connected(p2, b2).
```

4. Validation and verification

Verification and validation take place at all levels of design. At the highest level of abstraction, we validate the behavior of the abstract model against the requirements. We can also assert properties that a valid specification should possess, and attempt to verify these with informal reasoning or formal proofs. At lower levels, we may additionally verify invariants that show how higher-level abstractions are correctly implemented using objects that are closer to the implementation level.

4.1 Validation

Abstractions are particularly useful when the development involves people with no computing background. High-level operations performed by abstract objects enable such participants to use familiar concepts in validation, instead of using concepts arising from available implementation techniques.

If we only use implementation-level entities in validation, the essence of behavior is obscured behind implementation details. On the other hand, validating high-level abstractions whose behavior is not well defined is of little use. A combination of suitable abstractions and a well defined language for expressing them are thus needed.

The DisCo language has an operational interpretation, so an instance of the specification may be animated with the DisCo animation tool (Systä, 1991). Animation is particularly useful for embedded systems, where the behavior of the system as a whole is essential. Industrial experience also supports the use of animations (Isojärvi, 1997).

4.2 Verification

In addition to validating the behavior of a high level specification, we may also formulate properties that should hold if the specification correctly reflects our requirements. For example, we might want each phone to be involved in at most one call at a time, formulated as

forall c1, c2 **in** call, p **in** phone:
  p **in** c1.parties **and** p **in** c2.parties
    implies c1 = c2.

If this assertion is included in the specification, the animation tool evaluates the assertion after each step, and informs the user in case of a violation. This helps in catching errors when a specification is being developed.

Since the language has a precise semantics, we can also verify properties either informally or formally. In simple cases informal reasoning may be sufficient, but if more assurance is desired, the specification may be mechanically mapped to the logic of the PVS (Owre *et al.*, 1992) theorem prover, where mechanical formal verification may be carried out (Kellomäki, 1997). The assertions presented in this paper have been verified in this manner.

The DisCo specification methodology encourages the use of abstractions that are later implemented with lower level entities. This incurs an obligation to show that the abstractions are correctly implemented. These proof obligations are invariants linking abstract and concrete variables, and they can be verified with the desired level of formality.

5. Related work

In (Awad *et al.*, 1997), Awad and Ziegler recognize the need for representing collective behavior explicitly. They propose using Statecharts of a 'floating nature' to

capture aspects of the behavior that cuts across objects and classes. This is very close to our approach. However, unlike them, we use a language with a formal basis which makes it possible to use the abstraction for early validation and verification.

Increasing use of design patterns (Gamma et al., 1995), (Buschmann et al., 1996) can be interpreted to bear goals similar to the discussed approach. The development is initiated by using abstractions of inter-object cooperation, referred to by using the name of the pattern. However, unlike in pattern-oriented development, we have a well-defined state for the abstractions as well, providing enhanced validation and verification facilities.

6. Conclusions

We presented an approach for modeling distributed state as an object. Despite being an abstraction, such an object can bear significant information for early validation and formal verification. Thus, use of high-level abstractions results in an increased confidence that the resulting system will be satisfying.

Acknowledgments

This research has been partly supported by the Academy of Finland (project 40500).

References

Awad, M. and Ziegler, J. (1997). A pracatical approach to object-oriented state modeling. *Software - Practice and Experience*, 27(3):311-328, Mar. 1997.

Back, R.J.R and Kurki-Suonio, R. (1988). Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, (10):513-445.

Buschmann, F., Meunier, R., Rohneert, H., Sommerlad, P., and Stal, M. (1996) *A System of Patterns.* John Wiley & Sons.

Durr, E.H. and van Katwijk, J. (1992). VDM++ - A formal specification language for object-oriented designs. In *Technology of Object-oriented Languages and Systems*, Prentice-Hall International, pages 63-78. Proceedings of Tools Europe '92.

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

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231-274, June 1987.

Isojärvi, S. (1997). DisCo and Nokia: Experiences of DisCo with modeling real-time system in multiprocessor environment. FMEIndSem'97, Otaniemi, Finland, February 20, 1997.

Järvinen, H.-M. and Kurki-Suonio, R. (1991). DisCo specification language: marriage of actions and objects. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, IEEE Computer Society Press, pages 142-151.

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, pages 63-71.

Kellomäki, P. (1997). Verification of reactive systems using DisCo and PVS. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *FME'97: Industrial Applications and Strengthened*

Foundations of Formal Methods, number 1313 in Lecture Notes in Computer Science, Springer-Verlag, pages 589-604.

Kurki-Suonio, R. and Mikkonen, T. (1997). Liberating object-oriented modeling from programming-level abstractions. In J. Bosch and S. Mitchell, editors, *Object-Oriented Technology: ECOOP'97 Workshop Reader*, number 1357 in Lecture Notes in Computer Science, Springer-Verlag, pages 195-199.

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

Lano, K.C. (1991). Z++, an object-orientated extension to Z. In J.E. Nicholls, editor, Z User Workshop, Oxford 1990, Workshops in Computing, Springer-Verlag, pages 151-172.

Owre, S., Rushby, J.M., and Shankar, N. (1992). PVS: A prototype verification system. In D. Kapur, editor, 11th International Conference on Automated Deduction, number 607 in Lecture Notes in Artificial Intelligence, Springer-Verlag, pages 748-752.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object-Oriented Modeling and Design*. Prentice Hall.

Systä, K. (1991). A graphical tool for specification of reactive systems. In *Proceedings of the Euromicro'91 Workshop on Real-Time Systems*, IEEE Computer Society Press, pages 12-19.

Weiser, M. (1982). Programmers use slices when debugging. *Communication of the ACM*, 25(7):446-452, July 1982.