Note that our point in this paper is not to discuss the characteristics of synchronization schemes with growing complexity but in showing how our framework helps at reusing and customizing various levels of synchronization schemes, in order to produce (and implement) such refinements and variants. (You may again look at Fig. 2 which summarizes the hierarchy of synchronization schemes/classes that have been partially developed in this section.)

6 Evaluation, Related and Future Work

6.1 Evaluation of Current Framework

As we shown, implementations of various languages models, constructs and synchronization schemes have been rather easily integrated within a single framework. The benefits are in the possibilities to reuse and refine such models, constructs and schemes and to actually compare, apply and test them on real programs developed from the standard Smalltalk-80 environment. It is also possible to select and apply various synchronization schemes (and generally speaking various OOCP models) locally to various parts and computational contexts of a whole program/system.

Meanwhile, this experience in developing a hierarchy raises the general methodological issue of how to best extend such a taxonomy of classes while maintaining it highly modular. Currently this is the sole responsibility of the designer and implementor. As we have now some significant number of classes within the three main components taxonomies, it will be interesting to see how semi-automatic hierarchy management/reclassification mechanisms (such as [Gas 92]) may help us to manage the evolution and refinement of the taxonomies.

6.2 Evaluation of Evolution

Note that the initial version of Actalk in 1988 was much smaller than current framework. The 88 kernel included only two component classes (activity and behavior were not separated yet) and much less methods. In 1994 we developed a large number of class extensions of the kernel to implement numerous OOCP computation models, languages and synchronization schemes. Current Actalk framework emerged through the identification of the stable parts (kernel components and their basic methods) and their parametrization (parameter methods). This complexified the architecture albeit increased the possibilities. By comparing initial platform and current framework, we note that the initial platform got the larger use by numerous teams who used the kernel as a basis for implementing systems based on active objects. Thus, by using terms defined by Christopher Alexander [Ale 75] and highlighted by Richard Gabriel [Gab 96], we could say that although the 94 framework provided more extensive and systematic modeling abilities, habitability and piecemeal growth was better achieved for the first, lighter version of Actalk.

6.3 Related Work

The pros of building object-oriented (and more specifically Smalltalk-based) generic platforms for classifying various programming constructs has also been demonstrated by other platforms, such as Simtalk (for modeling various simulation schemes [Bz 87]) as well as others (Classstalk, Prototalk...). A generic scheduler has also been developed, as part of the Actalk project, by Loïc Lescaudron to classify and parameterize various scheduling policies [LB 91, Bu 93].

Alternatives to represent various OOPC designs are some more formal approaches, as the object calculus proposed by Oscar Nierstrasz [Nie 93]. Note that our pragmatic approach allows experiments with actual programs within a sophisticated programming environment (Smalltalk-80 based) to help at developing and monitoring them [LB 91].

The component-based architecture of an Actalk active object is close to the component-based meta-architecture of CodA, designed by Jeff McAffer [Mca 95]. CodA provides finer grain components and more refined interface between them, but at the cost of more complexity.

There are also other alternatives to components and methods decomposition, as for instance with parameterizing the invocation of some methods via arguments, as advocated by the Hermes/ST architecture [FHR 93]. This is also related to the issue of having a minimal kernel and extending it with more complex protocols and mechanisms versus having a bigger kernel offering more protocols not necessarily all used at the kernel level (see also previous discussion in Sect. 6.2). In the case of Actalk, the kernel has been designed a while ago to be simple and efficient, as its initial motivation was mainly didactic [Bn 89]. The scope of Actalk has also been restricted as it does not address distribution aspects (as for CodA) or fault-tolerance aspects (as for Hermes/ST) but concentrates on OOPC language design.

6.4 Future Work

Besides the areas and directions for future work already mentioned, an important and general issue is the combination of various aspects/descriptions of a computational behavior. The decomposition of the Actalk architecture into orthogonal components, and their further decomposition in parameter methods, help at such combination. Meanwhile, it reaches some limitations when combining different versions of a same component (in the context of this paper, the activity/synchronization component). The programmer may have then to rely on some amount of explicit composition. A finer grain decomposition of components, as proposed by CodA [Mca 95], brings more independence and modularity, but it also still relies on a single component to cover activity and
synchronization concerns. Our belief is that it is difficult anyway to further decompose a complex aspect, such as activity/synchronization, in fully orthogonal pieces. (In other words, we ultimately reach some atoms or even quarks.) Therefore, we believe that we cannot avoid the general problem of composing non-fully orthogonal components, and that we should develop some rationale and methodology for doing so. Some starting propositions may for instance be found in [MCA 95].

7 Conclusion

In this paper we have described the design of a framework for object-oriented concurrent programming, its architecture, and examples of use. We implemented successive refinements and combinations of synchronization schemes in order to show the expressive power of our platform. We finally evaluated our experience, related it to other works, and pointed future areas of investigation.


References


[FHR93] M. PIAZZOLE, B.G. HUMM, and R.D. RANSON, "Object-Oriented Extendibility in Hermes/ST, a Transactional Distributed Programming Environment," Proc. of the ECOOP'93 Workshop on Object-


[GHJ+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Design, Addison-Wesley, 1995.


