A NEW PARADIGM FOR THE DESIGN OF CONCURRENT SYSTEMS

Ralph Back
Åbo Akademi, Fänriksgatan 3, SF-20500 Åbo

and

Reino Kurki-Suonio
Tampere University of Technology, P.O.Box 527, SF-33101 Tampere

ABSTRACT

A concurrent system is usually understood as a collection of *processes* that interact through some *communication mechanisms*. The active components of such a system are the processes; their interaction is described in terms of messages and operations on shared memory. The principal process communication mechanism of Ada, the *rendezvous*, is a high-level mechanism based on synchronizing messages.

In this position paper we argue that appropriate design methods and languages for real-time systems should be based on a notion of interaction that is compatible with the corresponding concepts in the implementation language but provides a higher level of abstraction. It is claimed that existing methods do not satisfy this requirement with respect to Ada. A new paradigm is therefore suggested that reverses the traditional setting as follows. The active components of the system are not the processes but the *interactions* in which they participate. Such interactions, called *joint actions*, can be executed whenever they are *enabled*, i.e. when the required processes are free and willing to participate in these actions.

This change of viewpoint has an effect on the kind of entities that are described and refined in the design process. In particular, refinement of joint actions changes the *granularity* of the system. i.e. the *atomicity* of events being considered. Another novelty introduced by joint actions is the application of the *production language* paradigm to the design of concurrent systems. As joint actions are a generalization of the rendezvous, Ada is a suitable implementation language for this approach.

INTRODUCTION

There is no sharp distinction between languages for *specification, design* and *implementation* of concurrent systems. *Precision* and *formality* are required in all stages. *Incompleteness* of the description should not increase when moving from specification towards implementation. Similarly, the *level of abstraction* should not get higher in this process. In addition, the corresponding notions at different levels of abstraction should be *compatible.*

The *rendezvous* concept in Ada is an advanced mechanism for process interaction. It raises two questions:

(1) Is it an appropriate notion for real-time and other concurrent systems?

(2) How should it be matched in specification and design languages?

In this paper we are not going to discuss (1); we accept that the answer is a qualified "yes", and that the significance of the qualifications can be reduced by suitable optimization techniques.

With respect to (2) we claim that existing notions, like Petri nets, signals, semaphores, critical regions, monitors, path expressions, and message queues, are either incompatible with the rendezvous or at a lower level of abstraction. Therefore, these do not provide a suitable basis for the treatment of concurrency in the design methods and languages to be used in connection with Ada.

JOINT ACTIONS

The notion we propose is called *joint actions*. It arose in connection with using the simple rendezvous mechanism of CSP to express some distributed algorithms [1, 2]. The key observation was that the focus of the design could be turned from the behavior of individual processes to the intended purpose of their interactions. In the example cases it was found useful to turn the traditional viewpoint upside down: processes are passive carriers of their local data, and their interactions are the active components of the system. In our model, all data are assumed to be local to some process, and the concept of shared memory is not utilized. (This does not preclude the use of shared memory in the implementation.)

More specifically, joint actions can be outlined as follows. For each joint action a there is a non-empty set of processes that are required for its execution. There is also a *guard*, g_a, which determines whether a is *enabled* or not. The *execution* of a is triggered when the guard g_a is true and none of the required processes is involved in another joint action. The *body* of a is then executed; this is a finite operation on the local data of the participating processes. A *private* action by a single process is a special case of a joint action.

An important special case from the implementation point of view is a *separable* guard. This is one that depends only on the local data of the participating processes and consists of components that can be evaluated by them in a distributed way. This kind of joint action system can be efficiently executed on distributed systems.

An implementation of a system of joint actions contains two kinds of concurrency. Firstly, a single joint action may involve concurrent operation of the participating processes. Secondly, several joint actions involving disjoint sets of processes may execute in parallel.

The associated model of computation is quite simple, as the processes behave like passive resources for the joint actions. This is an acceptable view for a design language; for an implementation language that would be doubtful. For purposes of reasoning it is important that a concurrent computation is always equivalent to a *sequential* execution of joint actions. Problems of concurrency are therefore reduced to those of *nondeterministic* sequential computations. The theoretical properties of the model of computation, and distributed implementation of joint action systems have been analysed in [3, 4, 5].

RELATIONSHIP WITH RENDEZVOUS

Rendezvous and joint actions are both based on synchronous process communication. Joint actions provide a higher level of abstraction and are therefore a possible match for the rendezvous in a specification or design language.

Although the body of an Ada rendezvous belongs to one of the tasks (processes) involved, it can be understood as an action carried out jointly by the caller and the callee. Its interpretation in terms of a joint action means, however, that also the subsequent private codes in the two tasks be included in the same joint action. For a real-time system this seems natural, as these private codes carry out the operations for which the communication probably was intended.

In its general form the notion of joint actions can be seen as an extension of the rendezvous: it is not restricted to two-process operations, and the guards are much more general than what can be directly implemented in Ada.

IMPACT ON DESIGN METHODS

In its full generality, the concept of joint actions is not suitable for an implementation language. The view we have is that the design of a real-time system is first given in terms of general joint actions, and is then *transformed stepwise* into a form that can be mechanically translated into the implementation language. The common base of synchronous communication makes Ada a suitable target for this process.

Finally, we give a few comments about the design method we envisage. Work is in progress to test these ideas with realistic case studies of real-time systems, and to develop tools to support such a design process.

The *processes* of a system are determined at a relatively early stage. *Object-oriented design* and the *physical location* of the data (in a distributed system) give some guidance here: processes are carriers of their local data in the first place. However, susequent transformations may *combine* several processes (for efficiency), or introduce *additional* ones (interface tasks).

The initial design is easier to verify if it uses *multiprocess actions* with *non-separable guards*. Transformations that lead from non-separable to separable guards have been considered in [1]. Usually such a process requires introduction of auxiliary actions. Problems of splitting complex actions into simpler ones, and the connections of this process to the logics of knowledge have been discussed in [6]. In general, none of such transformations is trivial, but tools can be developed to support them and to check whether mechanical translation into the target language can be carried out.

An essential property of the design process is that the granularity of atomic events is gradually refined, while some of the parallelism is introduced that is initially hidden. This possibility for *refinement of atomicity* should be considered very important in any design method for real-time systems: the most intricate problems often relate to the fine degree of atomicity in the implementation.

Joint actions can also be seen as a *production language* for distributed systems. This extends the application of the production language paradigm to the design of concurrent processes and real-time systems.

REFERENCES

1. R. J. R. Back and R. Kurki-Suonio, "Decentralization of process nets with a centralized control". *2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 131-142, August 1983.
2. --, "A case study in constructing distributed algorithms: distributed exchange sort". *Proceedings of the Winter School on Theoretical Computer Science*, pp. 1-33. Finnish Society of Information Processing Science, January 1984.
3. --, "Co-operation in distributed systems using symmetric multi-process handshaking". Åbo Akademi, Department of Information Processing, Report A 34, 1984.
4. --, "Serializability in distributed systems with handshaking". Department of Computer Science, Carnegie-Mellon University, Report CMU-CS-85-109, 1985.
5. R. J. R. Back, E. Hartikainen, and R. Kurki-Suonio, "Multi-process handshaking on broadcasting networks". Åbo Akademi, Department of Information Processing, Report A 42, 1985.
6. R. Kurki-Suonio, "Towards programming with knowledge expressions". *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, pp. 140-149. Association for Computing Machinery, January 1986.