On Structuring Cooperative and Competitive Concurrent Systems

Alexander Romanovsky

TECHNICAL REPORT SERIES

No. CS-TR-682
October 1999

Contact:
alexander.romanovsky@ncl.ac.uk
www.cs.ncl.ac.uk/people/alexander.romanovsky/home.formal

On Structuring Cooperative and Competitive Concurrent Systems

A. Romanovsky

Department of Computing Science

University of Newcastle upon Tyne

Developing advanced structuring techniques has always been of great importance for computer science and practice. Many structuring approaches are used to help capture certain characteristics of applications: group communications, replication features, file services, etc. Procedures were among the first general techniques intended for structuring application software. They reflect both the static and dynamic structures of sequential systems (a stack of procedure contexts of nested calls represents the state of program execution). The situation is much more complex in concurrent systems, in which the states of several concurrent components should be taken into consideration while describing the system behaviour. The purpose of this survey is to outline recent trends in developing structuring approaches for competitive and cooperative concurrent systems and to discuss different directions of research in this area and their interrelations.

Keywords: complex concurrent systems, fault tolerance, atomic actions, conversations, atomic transactions, coordinated atomic actions

1. Competitive and Cooperative Concurrent Systems

Concurrent systems are difficult to design, to understand and to analyse. Researchers usually work in two complementary directions: developing services for particular application areas (or in order to provide a particular functionality) and identifying general approaches/techniques. General classifications of concurrent systems have always played an important role in the latter as they make it possible to concentrate on characteristics which are specific for different categories of systems and to develop methodologies and supports which facilitate developing systems of different categories. In this paper we will use a generalised classification of concurrent systems that is arrived at in [1] (which, in its turn, follows classifications in [2, 3]). Three categories are outlined here; they are independent, or disjoint, competing and cooperating systems. Many researchers find this classification useful and adhere to it in their work [1, 4, 5].

We assume that systems consist of components which can be either *active*, so that they are associated with (represent) system execution (e.g. processes, active objects, tasks, threads, clients, users) or *passive*, in which case they do not have their own activity and are used by the active components (e.g. objects, modules, instances of abstract data types, system resources, files, servers). A system is concurrent when it contains more than one active component.

Competitive concurrency exists when two or more active components are designed separately, are not aware of each other, but use the same passive components. The former have to compete for the latter and keep them at their disposal until there is no more need in them. Normally, components compete for a resource which knows nothing about the components that can use it. It can serve any clients if it is not busy. For example, competitive concurrency exists in client/server systems, in databases, in operating systems when tasks use the same system resources (say, heap or printers), in concurrent object-oriented systems when several threads access the same object.

Cooperative concurrency exists when several components cooperate, i.e. do some job together and are aware of this. They can communicate by resource sharing or explicitly, but the important thing is that they have been designed together so that they would cooperate to achieve their joint goal and use each other's help and results. They synchronise their execution and can wait for information computed by another cooperating component. This is a joint activity of several concurrent components with equal rights which are aware of each other. In order to cooperate, they have to share knowledge (of a name) which would be representative of their cooperation. This can be each other's names, the name of the information that they exchange, the name of a shared resource, or of cooperation. Some examples of such systems are: real time applications, control systems, CSCW systems; also, systems executing parallel computations and systolic algorithms, sharing a job, reaching agreement, electing a leader.

C.A.R. Hoare [3] explains that while disjoint processes work on disjoint data spaces, competing ones work on the same data, but it is guaranteed that this is done as if each of the competing processes had these data at its sole disposal. Processes cooperate if they update common variables by commutative operations and in a disciplined way (to guarantee a consistent access). A similar classification is used in [2]. The authors describe the differences between cooperation and competition in the following way. Cooperation includes all interactions which are anticipated and desired. Competition includes those which, though anticipated and acceptable, are undesirable, and usually involves access to resources that must be used serially by different processes.

This categorisation of systems is not explicitly related to the way in which components exchange information or synchronise their execution [6]. Features supporting these are of a low level and, generally speaking, one can build either cooperative or competitive systems using nearly the same synchronisation features, although some of them are more commonly used for building systems of a particular category. For example, RPCs, object method calls, shared variables and monitors are often used by active components in competitive systems to access the same passive components; equally, they can be used in cooperative systems by two clients to cooperate via servers. Symmetric rendez-vous, message exchange with known senders and receivers, signals are more suitable for programming cooperative relations. A typical grouping of types of concurrency into message-oriented and object-oriented ones is very similar to the general categorisation of concurrent systems discussed above; for example, protected objects were introduced into Ada 95 to extend the language by data-based concurrency as opposed to Ada 83 message-based concurrency which relies on inter-task rendez-vous [7]. One purpose of this, we believe, was to facilitate the design of competitive systems.

The two categories of concurrency correspond to two main styles of system design. The choice between competitive and cooperative concurrency is made by system designers. An identical or a similar system can be thought of differently and designed using either concurrency. This was first reported in [8] where the two main approaches (called message-oriented and procedure-oriented) to building operating systems are identified; the two system models (process and object ones) and the two corresponding approaches to system structuring (conversations and atomic transactions) are outlined in [9].

2. System Structuring and Fault Tolerance

Choosing the most suitable ways for system structuring is vital for dealing with complexity. B. Randell explains this in the following way: "Complexity can be reduced significantly by ensuring that the system is constructed out of a well-chosen set of largely independent components, which interact in a well-understood way" [10]. From our point of view, it is important to differentiate between static and dynamic structures of systems. They are different and should both be kept in mind by system designers while developing the system. This is why the complexity of both should be reduced using some appropriate structuring techniques. In sequential programs, the static system structure consists of modules and procedures (which can have modules and procedures declared inside), the dynamic one is represented as a stack of nested block contexts. Generally speaking, concurrent systems can be described statically as a set of passive and active components. In the rest of the paper we will concentrate on dynamic techniques developed for structuring concurrent systems. Designing concurrent systems using these techniques are important in many respects. Most significantly, it helps designers capture complex system behaviour and imposes a disciplined structured approach to achieving system fault tolerance on them.

It is only natural that all structuring techniques are associated with a fault tolerant mechanism because the units of the dynamic system structure are thereby easily related to all phases of providing fault tolerance [1]: they confine the damage, serve as areas of the error propagation, recovery features can be associated with them, etc.; thus providing fault tolerance is radically simplified. One cannot write a program that will tolerate any possible fault differently depending on the peculiarities of each point of the concurrent system execution where it can be detected. A certain unit of identical fault tolerance behaviour (error detecting and recovering) should be introduced. This is the underlying principle of all existing fault tolerance schemes (e.g. exceptions, rollback) [1]. Depending on fault assumptions, different fault tolerance features can be associated with structuring units; they usually rely on one of the two main classes of recovery [1]: backward (rolling all components participating in the faulty unit back to the previous correct state) and forward (recovering the unit participants into a correct state) error recovery. The former uses either diversely implemented software or simple retry; the latter is usually application-specific and relies on an exception handling mechanism.

Techniques which are used for structuring competitive and cooperative systems are different but the general idea is to make the execution of these units of system structure atomic, indivisible and invisible. This provides a very effective way of reducing complexity because the execution of all nested units is hidden on the level of the containing unit. Atomicity in the fault tolerance context usually assumes all-or-nothing semantics: if a fault has happened inside a unit and it was not possible to tolerate it transparently, then it is guaranteed that the containing unit is informed about it and that it is in a state in which it would be if the execution of the failed unit had not started. The execution of nested and sibling units is atomic for any containing unit (i.e. for any of its participants) and for any components outside this unit.

One of the most important properties of any system structuring technique must be its ability to be applied recursively while constructing the system [11]. Without this its applicability is limited. A well-known approach which is oriented towards developing complex applications involves nesting the units of the dynamic system structure. The rules of nesting are usually simple [12, 13]: any nested unit must be completed before the containing unit can be completed; the execution of the nested unit is indivisible and invisible for the containing and for the sibling ones; the nested unit results cannot be seen (are not committed) outside the containing unit until this containing unit is completed; only components taking part in the containing unit can participate in the nested ones.

Formal specification, verification and validation of concurrent systems which are structured of atomic units are essentially simplified. Many formal techniques can be applied only if some atomic units of execution are used and for many real applications this seems to be the best way of reducing complexity and making formal approaches feasible [14-17].

3. Structuring Techniques

3.1. Disjoint Systems

Understanding and reasoning about disjoint concurrent systems is simplified because the execution of an active component does not affect that of another one. Sequential structuring is used for these systems (nested procedures, blocks or method calls). Support is simple because processes do not work with the same passive components. Well-known approaches developed for sequential programs can be used to provide fault tolerance [1]: exception handling, recovery blocks, checkpointing, N-version programming. The system shown in Figure 1 has two disjoint processes, the execution of either of which is structured using a sequential approach; information does not cross the unit boundaries.

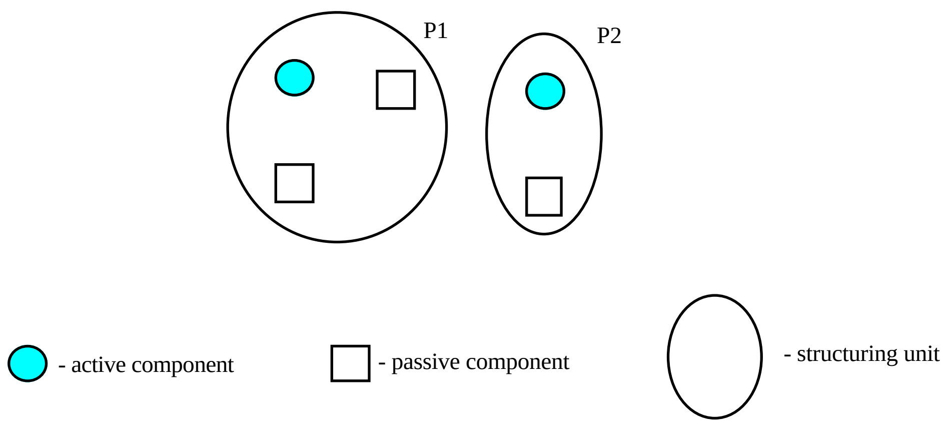

Figure 1. Two disjoint processes P1 and P2 accessing disjoint sets of passive components

3.2. Cooperative Systems: Conversations and Atomic Actions

There are two main structuring techniques for designing cooperative systems and for incorporating software fault tolerance into them in a disciplined way: conversations and atomic actions [1, 9, 12, 18]. They offer a highly efficient method of cooperative system design. Participants (active components) dynamically enter and leave a conversation or an atomic action (their exit has to be executed synchronously to guarantee the unit properties) and communicate within it in such a way that no information flow is allowed to cross its boundary. A nested unit includes only a subset of participants in the outermost one. The execution of each unit is atomic for outside components, which have no access to the intermediate states of participants. Participants are designed to cooperate and to be recovered within the unit. Each of these units encompasses several occurrences of communication between its participants, thus creating the level of cooperation in the system and serving as a building block of the cooperative system design. All this obviously restricts system design but makes it possible to regard each unit as a recovery region (beyond which erroneous information cannot be spread) and to attach fault tolerance features to each individual unit. Basically, these features provide error detection and recovery within units of cooperation: when an error has been detected, a corresponding recovery starts within the unit. With cooperative concurrency, one cannot rely on the recovery of just one component (even if just this one detects an error): all of them should be recovered and cooperative recovery should be provided. The reasons for this are well known: components freely exchange information inside the unit, so the potential damage area includes all of them; they are designed together to achieve their joint goal and they have to try and do this even if an error happens (their recovery has to be designed cooperatively); they can ensure the correct execution of the unit of cooperation only through a cooperative activity.

Conversations were the first structuring technique oriented towards developing cooperative systems. A basic description of a computational model of these units was discussed by B.Randell in [12]. This concept proved to be fundamental for all research in the field of software fault tolerating for concurrent cooperative systems. Logically, processes enter a conversation at the same time (in a distributed setting this means that entry events are concurrent); a recovery point is established in each of them (Figure 2). They freely exchange information within the conversation but cannot communicate with any outside process (in our model, any features of information exchange, e.g. messages, shared variables, are viewed as passive components). When all processes participating in the conversation have come to the end of the conversation, the acceptance test is to be checked. If it has been satisfied, the processes leave the conversation (thus, they can leave the conversation only at the same time). Otherwise they restore their states from recovery points. Should any process fail during the conversation execution, all the other processes are rolled back to their recovery points as well. Software diversity is used here: several alternates are to be implemented for each process. A second alternate is attempted in each process after state restoration. The conversation may be nested; in this case, a subset of processes from the containing conversation participate in it.

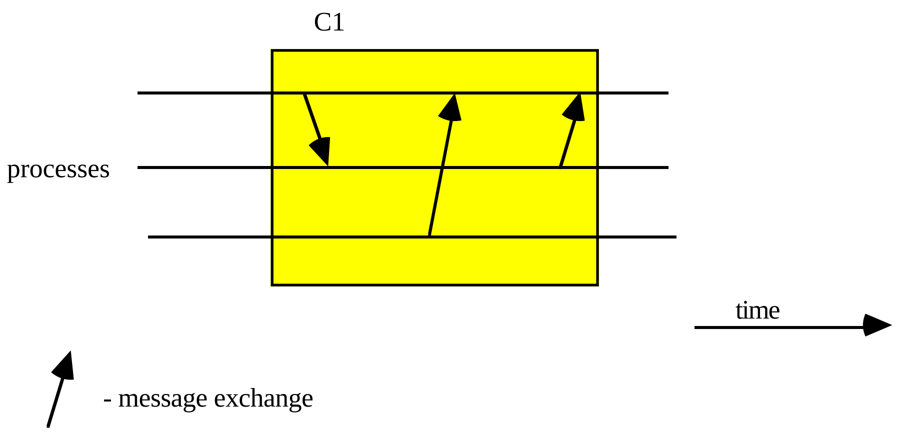

Figure 2. Conversation C1: three participants

Atomic actions were developed as a generalisation of conversations [18, 19]. They can use backward error recovery, forward error recovery or a combination of these. Backward error recovery, identical to the conversation recovery described above, does not depend on the application much and can be made transparent (or provided, to a considerable degree, by the action support) because it uses the rollback of all action participants to recover the system. Forward recovery usually relies on an exception mechanism and may incorporate an additional mechanism to resolve multiple exceptions raised concurrently in several participants. This can be done by designing a resolution tree to impose a partial order on all action exceptions in such a way that a higher level exception has a handler capable of handling any lower level exception. Exception handlers are attached to each action participant, and handlers for the same exception (the resolved one if several concurrent exceptions have been raised) are called in all of them (see Figure 3). This recovery is application-dependent by nature and this is why only basic support (though its implementation often requires serious effort) and a general structuring mechanism are provided by atomic actions. Note that we follow here the terminology in [18], and use the term "atomic actions" for units of cooperation; it is unfortunate that some researchers use it to refer to units of competitive system design.

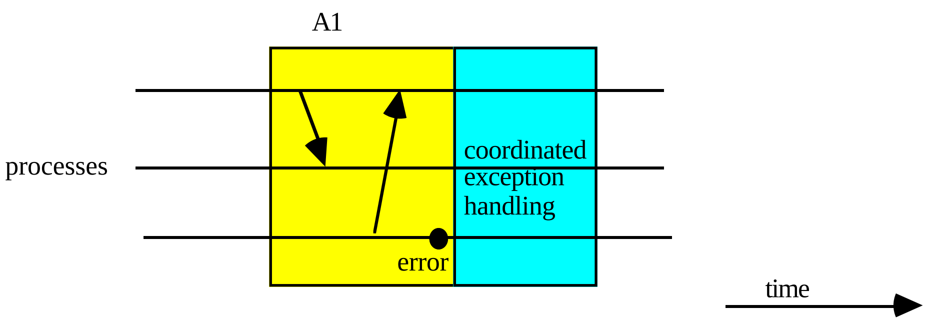

Figure 3. Atomic action A1: an error being detected, all three participants are involved in coordinated recovery (exception handling)

The execution of the system shown in Figure 4 consists of two atomic actions. Each of them has several processes which cooperate. Passive objects inside actions are both the means of this cooperation and data representing the action/system/process state. It is worth noting that the execution of these actions with respect to each other (as a whole) is disjoint. Generally speaking, the results of the action execution are represented by states of both passive and active components.

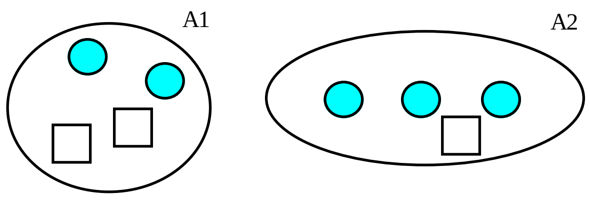

Figure 4. Two atomic actions (or conversations) A1 and A2 with two and three active participants respectively

3.3. Atomic Transactions

There are many types of structuring units developed for competitive systems. They come in different sorts, on different system levels, and are intended for different application areas. The first schemes were developed for shared resources in multi-user operating systems and databases. One of the first approaches describing general rules for structuring competitive applications and for introducing atomic units of execution into programming interface was the action procedure scheme by D.B. Lomet [20]. This paper shows, in particular, that systems become unnecessarily restrictive and complicated when monitors are used to guarantee execution atomicity. The proposed supporting mechanism guarantees that the same action procedure can be executed by several active participants (processes) with their concurrency maximised, so that all operations performed by a callee process inside a procedure on a set of objects (passive components) are seen by other active participants as if they were executed atomically. The recovery of each callee is isolated and relies on undoing all operations which have been performed inside the procedure before an error has been detected: this provides the nothing semantics in case of errors.

Atomic transactions are a technique which is most often used for structuring competitive systems [9, 21-23]. This is a very powerful approach which takes care of the following four properties of these structuring units: atomicity, consistency, isolation and durability (the ACID properties). Consistency means that the execution of any transaction on its own before completion is a correct transformation of the states of passive components and does not violate their integrity. The isolation (serializability) property plays a major role in providing competitiveness: the designer of a component which is going to use some resources does not have to know about other components competing for the same resources; this competition is hidden and does not affect the component design in any way. It is guaranteed that, even when several transactions are executed simultaneously, they do not affect each other, and the recovery of any of them is separated from the execution of the others. Durability is understood as the ability of the states of passive components to survive any assumed hardware faults which can happen after the top level transaction has been successfully completed (committed). The atomic transaction scheme relies on three standard operations: start, abort and commit transactions, which mark the boundaries of an atomic transaction (Figure 5). Within these boundaries, all passive components (resources) needed for a component are treated in a special way: they can be accessed by several transactions at the same time but the ACID properties of these transactions are guaranteed by transaction support. An atomic transaction encompasses several operations on a resource (passive component) and, in this sense, is a concept of a higher level than any individual operation (communication).

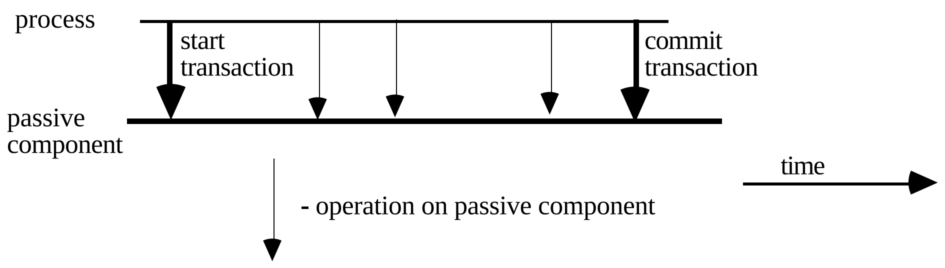

Figure 5. An atomic transaction is started by an active component that afterwards executes several operations on a passive component and commits the transaction

Recently the concept of multithreaded transactions [24] was developed to allow several active components (threads) to take part in the same transaction and to operate together on the same set of objects (passive components). Several multithreaded transactions can compete for the same objects. It is important to keep in mind that participants of the same transaction do not cooperate in any way (Figure 6): the scheme does not provide any features for them to do so, so that even if there is a need for them to cooperate, it is up to designers to program this correctly because it is not a part of the model. The execution of threads is not synchronised within a multithreaded transaction, so that, for example, some of them can leave it before the object state has been committed. There is no need to involve all threads into recovery because in this context recovery means transaction abort.

Atomic transactions, whatever their form, are intended for tolerating hardware faults only (this is the fault assumption in this model). Fault tolerance is provided by rolling back the states of objects (passive components). Atomic transactions do not incorporate any features for tolerating software or environmental faults or deal with the recovery of active components.

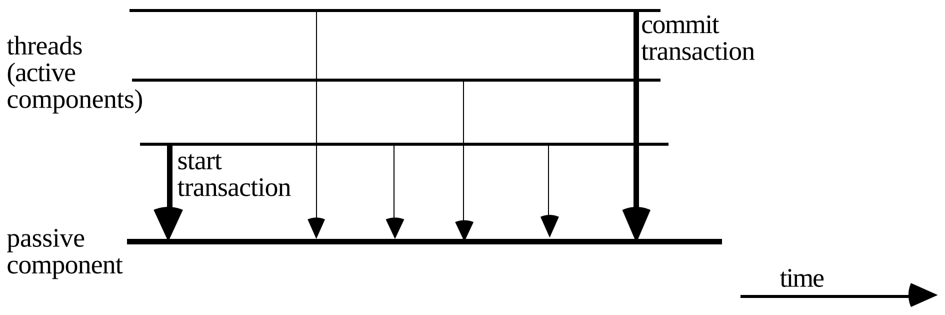

Figure 6. A multithreaded transaction is started by an active component; afterwards several of them perform operations on a passive component and one of them commits the transaction

The system in Figure 7 is structured as two atomic transactions which compete for two passive components, R1 and R2. Note that this paradigm does not include any means for process cooperation. The results of transaction execution are represented by states of passive components only. This technique guarantees that competing components (i.e. components from different transactions) are executed as if they were disjoint.

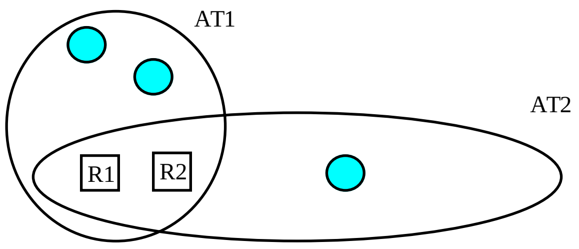

Figure 7. Two atomic transactions, AT1 and AT2, with two and one active participants respectively

There are always a lot of attempts made to extend the canonical transactional model by adding features which make this model more suitable for some practical applications (e.g. long-lived transactions, sagas, cooperative transactions, chain transactions, compensation transactions [21, 25]) by breaking the ACID properties in some particular ("controlled") way which better satisfies the needs of these applications [26]. It is usually not difficult to demonstrate that what all these applications require are additional features for cooperation and that each of these extensions introduces some kind of cooperation and an extended transactional support which controls this extension and provides some of the ACID properties "in spite" of this extension of the model.

3.4. Approaches Combining Cooperative and Competitive Concurrency

There are a lot of reasons why one may need a combination of atomic actions and atomic transactions. The world, as well as many realistic systems to be modelled/controlled by software, has elements of both cooperation and competition, and it is important to allow them to be combined within one system. One can make do with just one type of concurrency, but this may adversely affect performance and unnecessarily complicate the design. Another reason is that structuring units oriented towards cooperative and competitive concurrency will provide fault tolerance measures intended for tolerating different kinds of faults. In this section, we will discuss two recently proposed schemes which were developed to allow structuring complex systems that consist of cooperating and competing components.

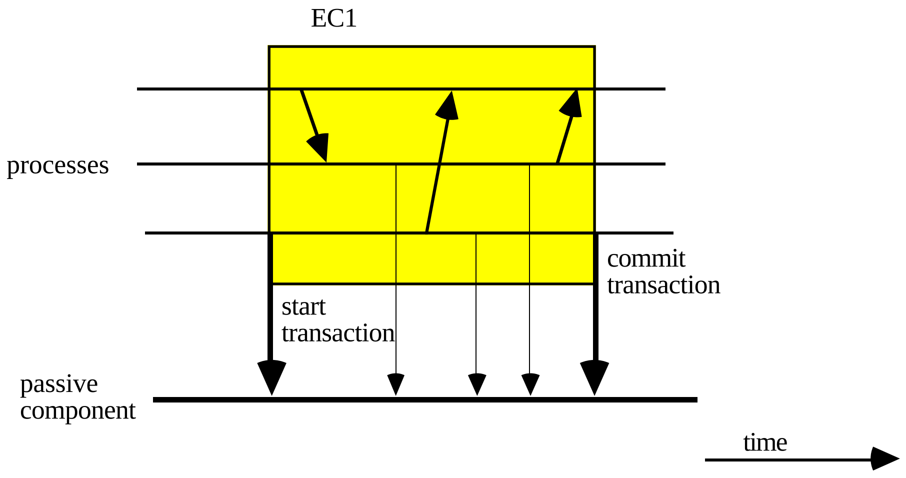

Figure 8. Extended conversation EC1 with three participants

The first successful attempt at combining cooperative and competitive concurrency was made by L. Strigini and F. Di Giandomenico [27]. We will refer to this scheme as "extended conversations". The heterogeneous systems under consideration can be made out of "conversational" cooperating components which synchronise their checkpointing/recovery via conversation-like mechanisms, and of server components which are supported by an atomic transaction mechanism; the latter accept requests (in the form of messages or remote procedure calls) to start, abort and commit transactions and serve them according to some policy that ensures proper concurrency control among competing transactions (see Figure 8). These systems are structured recursively using nested conversations which can access transactional objects (passive components). Extended conversational support guarantees that conversations correctly manipulate these objects. The support starts, aborts and commits (nested) transactions, transparently for the programmers of cooperating components, following strict rules which guarantee the correct behaviour of the composite system, and, in particular, provides a coordinated backward error recovery of all components involved in the extended conversation. In the papers which followed [28, 29], a complete description is given of algorithms which are executed programmer-transparently by this extended support to coordinate passive and active components; an implementation and a case study are discussed in details. This technique allows tolerating software faults of cooperating components. We can conclude by saying that extended conversations are intended for transparent coordination of heterogeneous system components which cooperate with the components for which they compete.

The Coordinated Atomic action (CA action) concept [5, 26] was introduced as a unified general approach to structuring complex concurrent activities and supporting error recovery between multiple interacting objects in a distributed object-oriented system. This paradigm provides a conceptual framework for dealing with both types of concurrency (cooperative and competitive) and achieving fault tolerance by extending and integrating two complementary concepts — atomic actions and transactions. CA actions have properties of both of them: atomic actions are used to control cooperative concurrency and to implement coordinated error recovery whilst transactions are used to maintain consistency of shared resources in the presence of failures and competitive concurrency. This allows tolerating faults of various types, as well as their combinations (using an extended resolution mechanism [26, 30]), occurring in different components involved in the CA action execution.

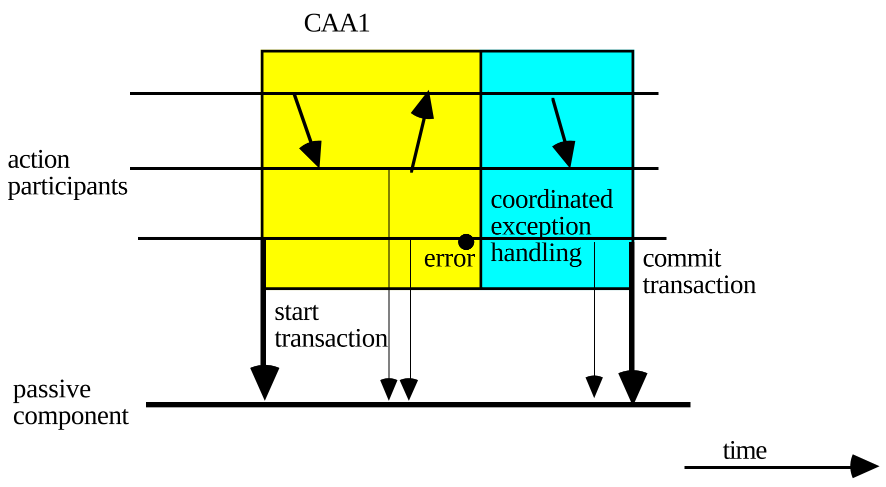

Figure 9. Coordinated Atomic action CAA1 with three participants which have executed coordinated forward error recovery

Each CA action is designed as a stylised multi-entry procedure with roles which are activated by action participants (active components) and which cooperate within the CA action (see Figure 9). Logically, the action starts when all roles have been activated and finishes when all of them reach the action end. The action can be completed either when no error has been detected or after successful recovery or when a failure exception has been propagated to the containing action. External (transactional) objects (passive components) can be used concurrently by several CA actions in such a way that information cannot be smuggled among them and that any sequence of operations on these objects bracketed by the CA action start and completion has the ACID properties with respect to other sequences. A CA action execution looks like an atomic transaction for the outside world. The state of the CA action is represented by a set of local and external objects; the CA action (either the action support or the application code) deals with these objects to guarantee their state restoration (which is vital primarily for backward error recovery). Participants can only cooperate (interact and coordinate their executions) through local objects.

CA actions are in many ways a generalisation of all structuring techniques discussed above. It is important that they allow us to use both cooperative and competitive concurrency in representing/modelling the system, to choose the right balance between cooperation and competition in the system and to apply the most suitable fault tolerance measures depending on the failure assumptions and resources available.

The execution of the system depicted in Figure 10 consists of two structuring units which compete for a passive component. Participants of these units cooperate through passive components belonging exclusively to one of these units. There are passive components of two kinds here: local (used for participant cooperation) and shared by several units (they have ACID properties). The results of the execution of these units are represented by the states of both active and passive components.

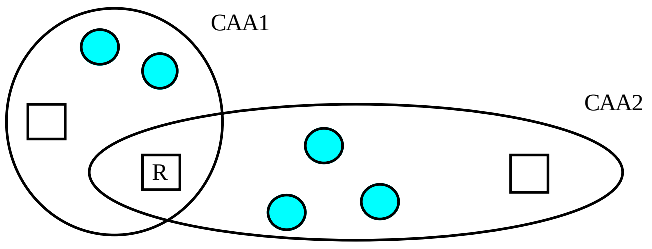

Figure 10. Two CA actions (can be extended conversations), CAA1 and CAA2, which are built out of cooperating components and which compete for passive component R

Several attempts have been made to introduce the CA action concept into earlier phases of the life cycle [31, 32]. This is important for many reasons and, in particular, because it would allow introducing both types of component relations into the system design and distinguishing between them as early as possible, and because this simplifies system modelling, formal reasoning about the system and transition between different phases of the life cycle. We expect that more work will be done in these directions. Hopefully, new structuring schemes based on CA actions will be developed in the future, which will allow a high level of flexibility and dynamicity in associating components with a particular type of concurrency (e.g. the same component will be able to be viewed as a passive one with respect to some components and an active one with respect to others), and new ways of system structuring will be proposed. For example, it will be possible to decompose both passive and active components into several components of different categories which either cooperate or compete and to use enhanced structuring techniques for such systems.

We would like to conclude by looking back on paper [33], which discusses problems of using conversations in production languages. The general authors' conclusion was that these languages are not suitable for programming conversations; to demonstrate this, they discuss difficulties in programming operations with servers within conversations. We believe that there is some misunderstanding here. For one thing, there are two types of concurrent systems: cooperating and competing ones. Different paradigms should be used to design them. Conversations are structuring units used to program cooperating systems; atomic transactions are units of which competing systems are built. Manipulating servers is obviously a competitive type of concurrency which should be programmed using transactions (but not conversations). A lot of the problems mentioned in [33] can be easily solved within the transactional paradigm: object and server sharing, action nesting, clients' anonymity, the absence of information smuggling. The schemes we have discussed are intended, in particular, for building systems in which conversations can access servers.

3.5. Summary

Table 1 summarises our considerations in Section 3.

technique	concurrency	faults	recovery
conversations (1976)	cooperative	design bugs	backward error recovery (software diversity)
atomic actions (1986)	cooperative	environmental faults, design bugs	forward error recovery (exceptions) and backward error recovery (software diversity)
atomic transactions (mid-60ies)	competitive	hardware faults	backward error recovery (retry)
extended conversations (1991)	cooperative and competitive	design bugs	backward error recovery (software diversity)
coordinated atomic actions (1995)	cooperative and competitive	environmental faults, design bugs, hardware faults	forward error recovery (exceptions) and backward error recovery (software diversity, retry)

Table 1. Comparison of different dynamic structuring techniques

4. Analysis

Atomic actions can be applied to structuring real time systems, process control, telephone switching systems and avionics [9] and achieving their fault tolerance. There have been several successful applications of the conversation and atomic action schemes to implementing practical systems, case studies and realistic examples: a naval command and control system [34], a simplified unmanned vehicle control system [35], a model of an antimissile defence C3 system [36], etc.; even then, the atomic transaction scheme is much wider spread in systems used in practice. Atomic transaction features have been introduced into many languages, programming, distributed and operating systems, etc. [21, 22]. For example, all database management systems heavily rely on the atomic transaction scheme.

It is interesting to try and analyse why the situation is so different for competitive and cooperative systems. There are many reasons for this. It is appreciated that we need concurrent systems, but programmers, developers and users seem to have got used to the idea that it is better to program a system and to work with it as if all resources belonged to it. This is competitive concurrency. We naturally prefer it this way because it facilitates our work and is much more comfortable. Supporting software (including the operating system and atomic transaction layers) guarantees that programmers developing software, software itself and users will work in their own environment without having to care about coordination and cooperation with the rest of the concurrent system. They can use an existing atomic transaction feature and assume this simplified model. This model suits many applications well. Atomic transactions dominate the world because it is easy to see the world in this way.

But it can be expensive and misleading if we try to use this model everywhere. Actually, the world is concurrent, distributed and cooperative, but no structuring techniques are widely used to program this in a disciplined way. It is deeply ironic that, to implement features supporting competitive units (atomic transactions) in modern distributed systems, programmers have to resort to using cooperation features. A certain inertia in education and in commonly used tools (existing on the market) which are mainly oriented towards the atomic transaction paradigm plays a role here. A lot of effort has been spent on developing/supporting elaborate transactional systems. It is difficult to change behaviour patterns.

There are historic reasons for this as well. Computing started with individual programmers having computers at their disposal. It was only natural that when first multi-user computers allowed several programmers to work at the same time, they created an impression that each of them had his/her own virtual computer: again, this idea fits into the competitive concurrency paradigm. There have been a lot of applications and design ideas which had their roots in competitive concurrency: databases, time sharing, client/server systems. But one would never think of programming any underlying services using atomic transactions. Concurrent activities might need to exchange information directly but not only through passive ACID components, they might need to synchronise their execution and have a joint goal. There are a lot of features for synchronising activities, and they are not part of the atomic transaction paradigm.

Another reason is the common belief that most errors are generated in hardware. It looks as if there were no software design faults. But the situation is just the opposite, and there is a lot of evidence that software faults and defects are becoming the dominating factor in decreasing the system dependability (see, for instance, [37-39]). It is getting clearer now that one needs structuring techniques which allow tolerating faults of all sorts and their combinations.

New complex distributed and concurrent systems (CSCW, Internet applications, advanced workflows, complex CAD, FMS and control systems, modern telephone switching systems, electricity grid control, multi-robot plants, complex middleware services, etc.) will require cooperation and, even more likely, some structuring schemes which combine cooperation and competition. The latter allow us to use both approaches to advantage; moreover, they do it in a systematic way which can be supported by standard services and layers. We believe that techniques which make it possible to structure complex concurrent systems with elements of cooperation and completion will become widely used in practice since they meet the requirements.

The authors of the extended conversation scheme [29] expect that their scheme can be of practical use as it is an important aspect of large, complex modern systems that they may include parts that belong to traditionally separate application domains, and thus are naturally designed following different approaches. For instance, the "intelligent network" evolution of the telephone network relies on the interaction of real-time control components with vast on-line databases, with different timing and availability requirements. Similar factors of heterogeneity may be expected in other large-scale control systems, e.g. in transportation (air traffic, vehicle-highway systems), and integrated manufacturing control.

Recently the CA action concept has been intensively applied to designing a series of safety-critical industry-oriented Production Cell case studies (with environmental and transient faults, real-time constraints, etc.) [17, 40, 41] and a non-conventional concurrent computational model (the Gamma computation) [42].

Introducing cooperation and software fault tolerance into atomic transactions in an ad hoc way seems to be a wrong and somehow dangerous approach. Our belief is that, although it is clearly possible to program all fault tolerant concurrent applications using atomic transactions (let alone the Turing Machine), the new structuring techniques discussed above will be more widely used in practice because a lot of complex applications will greatly benefit from using them.

It is our belief that developing and using modern dynamic structuring techniques is vital if we want to cope with the complexity and heterogeneity of applications that we are facing now. The two recently developed approaches (extended conversations and CA actions) appear to be generally applicable and capable of reducing system complexity and imposing a disciplined way of achieving fault tolerance on all levels of system design.

We are well aware that the general considerations and conclusions presented in the paper are abstract and sometimes cannot be easily related to a huge variety of existing systems and applications. We believe that, despite this, they should be useful for understanding general concepts and trends in this area.

Acknowledgements. First of all, my thanks go to Prof. B.Randell, working with whom has always been a great inspiration for me, and to my colleagues in ESPRIT DeVa LTR Project, ESPRIT PDCS BR Project and EPSRC/UK DISCS Project. I would also like to acknowledge the support from these Projects.

References:

[1] Lee, P.A. and Anderson, T. (1990) *Fault Tolerance: Principles and Practice*. Springer-Verlag, Wien - New York.

[2] Horning, J.J. and Randell, B. (1973) Process Structuring. *Computer Surveys*, 5, 1, 5-30.

[3] Hoare, C.A.R. (1976) Parallel Programming: an Axiomatic Approach. In G. Goos and J. Hartmaur, (eds), *Languages Hierarchies and Interfaces, Lecture Notes in Computer Science, LNCS-46*. Springer-Verlag, Berlin - New York.

[4] Burns, A. and Wellings, A. (1997) *Real-Time Systems and Programming Languages*. Addison Wesley, Harlow, England.

[5] Xu, J., *et al.* (1995) Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery, in: Proc. 25th Int'l Symp. on Fault-Tolerant Computing. IEEE CS Press, Pasadena, California, 499-508.

[6] Romanovsky, A. (1995) Software Diversity as a Way to Well-Structured Concurrent Software. *ACM Operating Systems Review*, 29, 3, 85-90.

[7] Burns, A. and Wellings, A. (1995) *Concurrency in Ada*. Cambridge University Press, Cambridge.

[8] Lauer, H.C. and Needham, R.M. (1979) On the Duality of Operating System Structures. *ACM Operating System Review*, 13, 2, 3-19.

[9] Shrivastava, S.K., Mancini, L.V. and Randell, B. (1993) The Duality of Fault-Tolerant System Structures. *Software - Practice and Experience*, 23, 7, 773-798.

[10] Randell, B. (1986) System Design and Structuring. *The Computer Journal*, 29, 4, 300-306.

[11] Randell, B. (1983) Recursive Structured Distributed Computing Systems, in: Proc. Third Symposium on Reliability in Distributed Software and Database Systems. Florida, USA, 3-11.

[12] Randell, B. (1975) System Structure for Software Fault Tolerance. *IEEE Trans. on Software Eng.*, SE-1, 2, 220-232.

[13] Moss, J.E.M. (1981) Nested Transactions: An Approach to Reliable Distributed Computing, Technical Report 260 (Ph.D. Thesis), MIT Lab. for Computer Science.

[14] Jalote, P. and Campbell, R. (1985) Atomic actions in concurrent systems, in: Proc. 5th Int'l Conf. on Distributed Computing Systems. IEEE CS Press, 184-191.

[15] Best, E. (1996) *Semantics of Sequential and Parallel Programming.* Prentice Hall, London -New York.

[16] Kurki-Suonio, R. and Mikkonen, T. (1998) Liberating object-oriented modeling from programming-level abstractions. In J. Bosch and S. Mitchell, (eds), *Object-Oriented Technology, LNCS 1357*. Springer-Verlag, Berlin - New York.

[17] Xu, J., *et al.* (1998) Rigorous Development of a Safety-Critical System Based on Coordinated Atomic Actions. DeVa ESPRIT LTR Project. Third year deliverable. (submitted to FTCS-29), LAAS, Toulouse, France.

[18] Campbell, R.H. and Randell, B. (1986) Error Recovery in Asynchronous Systems. *IEEE Trans. on Software Eng.*, SE-12, 8, 811-826.

[19] Randell, B. and Xu, J. (1995) The Evolution of the Recovery Blocks. In M.R. Lyu, (eds), *Software Fault Tolerance.* John Wiley & Sons, Chichester - New York.

[20] Lomet, L. (1979) Process Structuring, Synchronization, and Recovery Using Atomic Action. *ACM SIGPLAN Notices*, 12, 2, 128-137.

[21] Gray, J.N. and Reuter, A. (1993) *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, California.

[22] Lynch, N.A., Merrit, M., Weihl, W.E. and Fecete, A. (1993) *Atomic Transactions*. Morgan Kaufman, San Mateo, California.

[23] Weihl, W.E. and Liskov, B. (1985) Implementation of Resilient Atomic Data Types. *ACM Trans. on Programming Languages and Systems*, 7, 2, 244-269.

[24] Object Management Group (1996) Object Transaction Service. Draft 4, OMG Document, OMG.

[25] Elmagarmid, A.K. ed. (1993) *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, San Mateo, California.

[26] Randell, B., Romanovsky, A., Stroud, R.J., Xu, J. and Zorzo, A.F. (1997) Coordinated Atomic Actions: from Concept to Implementation, Technical Report 595, Computing Dept., University of Newcastle upon Tyne (http://www.cs.ncl.ac.uk/-research/trs/papers/595.ps).

[27] Strigini, L. and Di Giandomenico, F. (1991) Flexible schemes for application-level fault tolerance, in: Proc. 10th Int'l Symp. on Reliable Distributed Systems. IEEE CS Press, Pisa, Italy, 86-95.

[28] Strigini, L., Romanovsky, A. and Di Giandomenico, F. (1994) Recovery in heterogeneous systems, Technical Report 133, PDCS-2 ESPRIT Basic Research Project.

[29] Strigini, L., Di Giandomenico, F. and Romanovsky, A. (1997) Coordinated Backward Recovery between Client Processes and Data Servers. *IEE Proceedings on Software Engineering,* 144, 2, 134-146.

[30] Romanovsky, A., Xu, J. and Randell, B. (1996) Exception Handling and Resolution in Distributed Object-Oriented Systems, in: Proc. 16th Int'l Conf. on Distributed Computing Systems. IEEE CS Press, Hong Kong, 545-553.

[31] de Lemos, R. and Romanovsky, A. (1998) Coordinated Atomic Actions in Modelling Object Cooperation, in: Proc. 1st IEEE Int'l Symp. on Object-oriented Real-time Distributed Computing, ISORC'98. IEEE CS Press, Kyoto, Japan, 152-161.

[32] Di Marzo Serugendo, G., Guelfi, N., Romanovsky, A. and Zorzo, A.F. (1998) Formal Development and Validation of the DSGamma System Based on COOPN/2 and Coordinated Atomic Actions, Technical Report 98/265, Software Engineering Laboratory, Swiss Federal Institute of Technology Lausanne.

[33] Gregory, S.T. and Knight, J.C. (1989) On the Provision of Backward Error Recovery in Production Programming Languages, in: Proc. 19th Int'l Symp. on Fault-Tolerant Computing. IEEE CS Press, Chicago, Illinois, 507-511.

[34] Anderson, T., Barret, P.A., Halliwell, D.N. and Moulding, M.R. (1985) Software Fault Tolerance: an Evaluation. *IEEE Trans. on Software Eng.*, 11, 12, 1502-1510.

[35] Yang, S.M. and Kim, K.H. (1992) Implementation of the Conversation Scheme in Massage-Based Distributed Computer Systems. *IEEE TPDS*, 3, 5, 555-572.

[36] Kim, K.H. and Bacellar, L. (1997) Time-Bounded Cooperative Recovery with Distributed Real-Time Conversation Scheme, in: Proc. Third IEEE Workshop on Object-Oriented Real-time Dependable Systems, WORDS'97. IEEE, Newport Beach, California, 1-8.

[37] Laprie, J.-C. (1996) Software-based critical systems, in: Proc. 15th Int'l Conf. on Computer Safety, Reliability and Security. SAFECOMP'96. Vienna, Austria, 157-170.

[38] Lee, P.A. (1994) Software-Faults: The Remaining Problem in Fault Tolerant Systems? In M. Banatre and P.A. Lee, (eds), *Hardware and Software Architectures for Fault Tolerance. LNCS 774.* Springer-Verlag, Berlin - New York.

[39] Silva, J.G., Silva, L.M., Madeira, H. and Bernandino, J. (1994) A Fault-Tolerant Mechanism for Simple Controllers, in: Proc. EDCC-1. Springer-Verlag, Berlin, Germany, 39-55.

[40] Zorzo, F., *et al.* (1999) Using Coordinated Atomic Actions to Design Complex Safety-Critical Systems: The Production Cell Case Study. *Software - Practice and Experience (accepted).* Preliminary version: http://www.newcastle.research.ec.org/de-va/papers/37.ps

[41] Romanovsky, A., Xu, J. and Randell, B. (1998) Exception Handling in Object-Oriented Real-Time Distributed Systems, in: Proc. 1st Int'l Symp. on Object-oriented Real-time Distributed Computing, ISORC'98. IEEE CS Press, Kyoto, Japan, 32-42.

[42] Romanovsky, A. and Zorzo, A.F. (1999) Coordinated Atomic Actions as a Technique for Implementing Distributed Gamma Computation. *Journal of Systems Architecture (Special Issue on New Trends in Programming and Execution Models for Parallel Architectures, Heterogeneously Distributed Systems and Mobile Computing)*, 45, 1357-1374.