Designing Reliable Real-Time Concurrent Object-Oriented Software Systems

Alfredo Capozucca
Laboratory for Advanced Software Systems
University of Luxembourg
6, rue Richard Coudenhove-Kalergi
L-1359 Luxembourg-Kirchberg
alfredo.capozucca@uni.lu

Nicolas Guelfi
Laboratory for Advanced Software Systems
University of Luxembourg
6, rue Richard Coudenhove-Kalergi
L-1359 Luxembourg-Kirchberg
nicolas.guelfi@uni.lu

ABSTRACT

Coordinated Atomic Actions is a conceptual framework used to increase the reliability (by fault tolerance) of concurrent object-oriented software systems. An extension of this conceptual framework to support the modelling of real-time software systems has been proposed. In this work we present our proposal for improvements of this extension focusing on recovery process optimisation, non-determinism reduction and time-related constructs extension.

Categories and Subject Descriptors

C.4 [**PERFORMANCE OF SYSTEMS**]: Fault tolerance and Modelling techniques

General Terms

Design, Reliability

Keywords

Coordinated Atomic Actions, Real-Time, Fault Tolerance

1. MOTIVATION

Coordinated atomic actions (CaaFWrk) [4] is a fault tolerance conceptual framework used to increase the reliability of concurrent object-oriented (OO) software systems. The kernel of the CaaFWrk is an abstraction, which is defined as a generalised form of the *atomic action*[1] concept. Such abstraction (called CAA from now on) allows a set of concurrent processes to perform a group of operations on a collection of objects. A CAA acts also as a damage confinement area, since it constraints the spread of errors to its enclosing context. It is achieved by associating recovery procedures to each atomic logic unit. Exceptions and exception handling features are used to identify the presence of an error and also to eliminate it by putting in place one of the recovery procedures associated to the atomic logic being executed.

Concurrent OO software systems with high levels of reliability belong to the domain set for which the CaaFWrk was meant to be used as design tool. Real-time software systems are (either inherent or imposed) concurrent and very often have reliability requirements ([2], pages 7-12). Thus, these types of software systems are first-class candidates to be designed using the CaaFWrk. However, the timing requirements imposed by most real-time software systems cannot be modelled (or, at least, not easily) by the CaaFWrk as it is. Thus, extensions to make it available for designing real-time software systems are required. A first attempt was proposed by Romanovsky et al. in [1]. In such work the CaaFWrk was extended to allow timing constraints to be placed over a CAA abstraction. We claim that there are still some open issues in areas like timing constraints over roles, roles/CAAs interleaving and recovery scheme. The aim of our work then is twofold: to give a detailed description of these issues and propose ways to address them. An extended version of this paper can be found in [3].

2. ISSUES AND SOLUTIONS

Timing constraints over roles and Timed-CaaFWrk recovery semantics: the only timing constraint allowed by the Timed-CaaFWrk to be set on a role is the maximum execution time. This is to ensure that the statements within the role do not execute longer than such amount of time. We believe that to facilitate the design of a role (and also of the CAA which it belongs to) the same kind of timing constraints the Timed-CaaFWrk allows to set on a CAA (i.e. deadline, minimum and maximum delay and maximum elapsed time -MET) should be possible over a role.

Among them, deadline and MET represent useful resources for the designer in order to check how a role evolves with respect to a time reference. Thus, they can be used to establish internal time-checkpoints to monitor the progress of the role. A role overrunning its associated deadline/MET represents an exceptional situation where the overall CAA goal may not be met on time, except certain recovery action is applied. Following the Timed-CaaFWrk semantics, any recovery action implies to engage every CAA's role to handle the exceptional situation (which, in some cases, can be excessive). We propose to extend the Timed-CaaFWrk recovery semantics to allow an exceptional situation to be handled in the context of the same role where it has been detected. With this approach the recovery action to be ap-

[1]Abstraction that allows the execution of a set of operation to be seen as only one indivisible operation.

SAC '09, March 8-12, 2009, Honolulu, Hawaii, U.S.A.

plied when a role overruns certain timing constraint may be performed by the same role. An interesting extension to our proposal would be also to allow setting multiple deadlines inside the same role. In this case a timing block will be required to define the set of instruction to be executed in less than certain units of time. A proposal could be ***less****(timeExpr)*{...}***exceed***{...}, where the primitive *exceed* would (optionally) allow to specify those instruction to be executed in case of reaching the deadline (i.e. *timeExpr*).

Roles/CAAs interleaving: the roles executing in a CAA perform their tasks concurrently. If the CAA is properly designed, then the functional output (i.e. CAA's goal) will be the same regardless the internal behaviour of its roles. What varies considerably is the timing behaviour of each role depending of how they are interleaved. The same situation arises when multiple CAAs are executed concurrently. The timing behaviour of each CAA would vary depending how they are interleaved. In this manner, the non-determinism found within a software system designed by several CAAs executing concurrently (and within each CAA, too) is of special care when timing constraints are imposed over roles and/or CAAs. As said in [2], on page 465: *a real-time systems needs to restrict the non-determinism found within concurrent systems.* A way to restrict the non-determinism found in a concurrent software system is by using certain **scheduling** policy. In our case (communication among roles/CAAs by local/external objects) a possible solution could be to use any of the existing *priority ceiling protocols* (PCP). This type of protocol was originally designed for single processor systems as a modification of the priority inheritance protocol (PIP) and assumes a pre-emptive dispatching ([2] pp.492).

Integrating a PCP into the Timed-CaaFWrk implies:
let caa_1 be a CAA composed of n roles ($r_i, i = 1..n$), such that its roles exchange information among them by m local objects ($lo_j, j = 1..m$), and it accesses over r external objects ($eo_k, k = 1..r$)), which are resources shared with other CAAs (i.e. caa_l ($l = 2..w$)):
(1) assign an unique priority to each role r_i ($i = 1..n$) belonging to caa_1,
(2) assign a ceiling priority to each local object lo_j ($j = 1..m$) used in caa_1 to exchange information among its roles (the ceiling priority of the local object lo_j ($j = 1..m$) will be the maximum priority of the roles r_i ($i = 1..n$) that use it),
(3) assign an unique priority to each CAA caa_l ($l = 1..w$),
(4) assign a ceiling priority to each external object eo_k ($k = 1..r$) that is the maximum priority of the caa_l ($l = 1..w$) that use it. With this policy, a role/CAA has a dynamic priority that is the maximum of its own static priority and the ceiling values of any local/external object it has locked. It must be noticed that, on a single processor system, the simple fact of applying PCP, ensures the mutual exclusion required over each shared resources. Thus, no synchronisation primitive would be required to provide mutual exclusion to protect such shared resources. Under the same conditions (i.e. single processor system) it is not possible to reach a deadlock state, since the ceiling protocols are a form of deadlock prevention. However, in a distributed system, to ensure such properties the PCP must maintain a global view of the acquired shared resources, which may lead to high communication overhead.

Pre-emptive scheme: the current Timed-CaaFWrk uses a pre-emptive scheme for the recovery process, instead of a blocking scheme. According to the pre-emptive scheme a role has the rights to interrupt any other roles belonging to the same CAA when it has detected an exception. Let caa_1 and caa_2 be two different CAAs such that caa_2 is nested in caa_1, then according to the principles of the pre-emptive scheme, caa_2 can be interrupted by any of the roles belonging to caa_1 when an exception has been detected in one of them. The interruption of a nested CAA is such a way represents its abortion. It is argued in [1] that aborting the nested CAAs rather than waiting for its completion (as done according to the blocking scheme) speeds up the overall CAA recovery process, which is very important when CAAs have timing constraints. We claim that not always the fact of aborting a nested CAA will speed up the recovery process of the enclosing CAA where the exception takes place. Assuming that caa_2 has a deadline d and needs n time units to be aborted, every time it has been executing for more than $d - n$ time units it will be faster for caa_1 to wait for caa_2 completion rather than abort it.

The proposal then is to use a mix between a pre-emptive and blocking schemes: a role does not have the rights to interrupt the execution of any other roles when it detects an exception; instead, it notifies to the scheduler of such event. Once the scheduler gets the notification about the exception, it will decide (based on information provided at design time) which nested CAAs have to be aborted and which ones have to be kept running until the end. Notice that in this context, only value-related exceptions have to be notified to the scheduler since time-related exceptions are automatically detected by the scheduler.

3. CONCLUSION

The main goal of this work was to find open issues in the Timed-CaaFWrk and to propose a way to address each of them. The new Timed-CaaFWrk obtained as result of integrating the proposed solutions should have better acceptance when designing real-time software systems. Whether it covers all the needs and is desirable for constructing this kind of software system can only be determined from future practical experience.

4. REFERENCES

[1] A.Romanovsky, Xu, J., and Randell, B. Coordinated exception handling in real-time distributed object systems. *Computer Systems Science and Engineering 14*, 4 (1999), 197–208.

[2] Burns, A., and Wellings, A. J. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[3] Capozucca, A., and Guelfi, N. Designing Reliable Real-Time Concurrent Object-Oriented Software Systems using Coordinated Atomic Actions: first round. Tech. Rep. TR-LASSY-08-06, Laboratory for Advanced Software Systems, University of Luxembourg, 2008.

[4] Xu, J., Randell, B., Romanovsky, A., Rubira, C. M., Stroud, R. J., and Wu, Z. Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. *Proceedings of the 25 International Symposium on Fault-Tolerant Computing* (1995), 499–508.