Controlled Reversibility and Compensations*

Ivan Lanese[1], Claudio Antares Mezzina[2], and Jean-Bernard Stefani[3]

[1] Focus Team, University of Bologna/INRIA, Italy
lanese@cs.unibo.it
[2] SOA Unit, FBK, Trento, Italy
mezzina@fbk.eu
[3] INRIA Grenoble-Rhône-Alpes, France
jean-bernard.stefani@inria.fr

Abstract. In this paper we report the main ideas of an ongoing thread of research that aims at exploiting reversibility mechanisms to define programming abstractions for dependable distributed systems. In particular, we discuss the issues posed by concurrency in the definition of controlled forms of reversibility. We also discuss the need of introducing compensations to deal with irreversible actions and to avoid to repeat past errors.

1 Motivation

In this paper we report the main ideas of an ongoing thread of research that aims at exploiting reversibility mechanisms to define programming abstractions for dependable distributed systems. Many such abstractions have been proposed in the literature, concerning for instance exception handling, checkpointing, transactions and the like [6,11,12], and made available to programmers as language primitives, libraries or middleware functions. However these different proposals lack formal foundations and do not generally compose well. This raises the question of whether some unifying framework can be found to shed light on the relations among these apparently unrelated mechanisms. Clearly, a number of these mechanisms are based on some form of *undo*, allowing to annul the effect of actions that lead to an error. We thus ask the following question:

If we were able to undo every action in a distributed program execution, would we be able to understand and integrate those different mechanisms?

We started our endeavor in the framework of concurrency theory and in particular using process calculi, developing small reversible languages and trying to understand their properties and their expressive power.

Paper Outline. Section 2 recalls the main features of reversibility in a concurrency setting. Section 3 discusses and compares different mechanisms for controlling reversibility. Section 4 outlines some ideas for combining reversibility and compensations. Section 5 concludes the paper.

* This work has been partially supported by the French National Research Agency (ANR), project REVER n. ANR 11 INSE 007.

R. Glück and T. Yokoyama (Eds.): RC 2012, LNCS 7581, pp. 233–240, 2013.

2 Reversibility in a Concurrency Setting

The problem of understanding reversibility in a process calculus scenario had already been considered in the seminal paper [8], with motivations coming from computational biology, where systems are naturally reversible. In [8] a reversible variant of CCS has been proposed. A main achievement of that paper is the definition of *causal consistency*, a formal criterion for reversibility in concurrent systems. In a concurrency setting (and even more in a distributed one) there may be no clear understanding of which was the last action performed by the system, or which was the previous state. With causally consistent reversibility one moves back by undoing any action that could have been the last one, i.e. any action on which no other action depends. Essentially, actions are undone in reverse order with respect to forward execution, up to possible swaps in the order of execution of concurrent actions.

To better understand this crucial concept, consider the example in Figure 1(a). From state M there are two possible paths leading to state N, one executing first a and then b (on the left), and the other executing first b and then a (on the right). If the two actions a and b are concurrent, possibly executed by physically remote components, it may be difficult to distinguish the two computations. Thus one should be able to reverse any of the two executions by reversing the other, i.e. if the forward computation proceeds by executing first a and then b (double-pointed arrow in Figure 1(b)), not only undoing first b and then a, but also undoing first a and then b (wave arrow in Figure 1(b)) is a valid reverse computation.

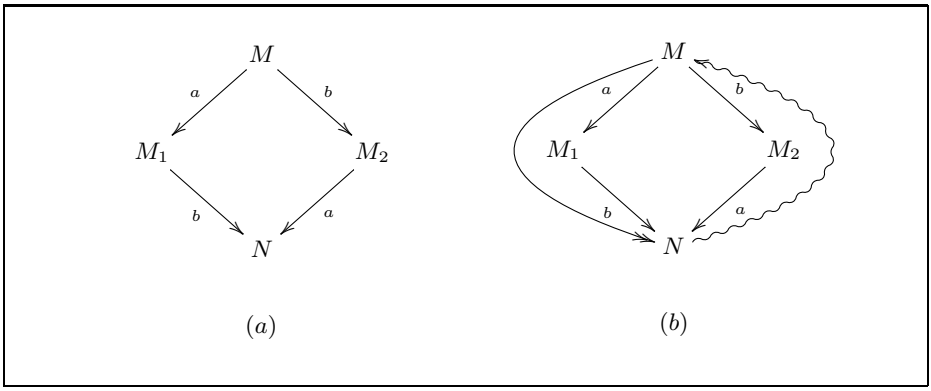

Fig. 1. Example of Causally Consistent Executions

To allow causally consistent reversibility one has to add history information to the different threads in a computation. Choosing threads as the granularity at which to store this information is suitable for causal consistency since actions inside the same thread are causally dependent, while actions in different threads

are mostly concurrent. Subsequent works have shown that such a framework can be applied to different process calculi, in particular to a family of CCS-like calculi [19], to the (higher-order) π-calculus [16], and to a subset of Oz [17].

3 Controlling Reversibility

In the works discussed above, reversibility is essentially *non-deterministic*, in the sense that, when both forward and backward steps are possible, there is no way of deciding whether to go forward or to go backward. This means that those works specify *how* the system can reverse a forward computation and what kind of information it should exploit, but they give no hint about *when* forward execution should be preferred over backward one and vice-versa.

In the fault-tolerance setting, there is a general answer to this question: to use reversibility for error recovery, the system should normally execute forward, and backward execution should be exploited only when needed to recover from errors. Such a general guideline, however, can be implemented using different strategies. We describe below three main strategies, specifying scenarios where they can be applied. This allows us to structure the design space of controlled reversibility, and to categorize the approaches in the literature accordingly.

Internal Control: Specific commands inside processes specify whether the process itself should go forward or backward. A possibility along this line has been explored in [9], where *irreversible* actions, i.e. actions that once performed cannot be undone, have been integrated in reversible CCS and shown able to implement a simple form of transactions. In [9], however, it is not clear how to relate irreversible actions and error recovery. To make this relation more apparent, we proposed a dual approach [15] where an explicit rollback primitive is used to trigger backward execution. The idea is that when an error is spotted, the rollback primitive can be used to go back to a consistent state. To specify how far back to go the rollback primitive takes as parameter a label referring to a past action, and it undoes all (and only) the actions causally dependent on it, that is all the actions generated because of it. This choice is coherent with, and indeed forced by, causally consistent reversibility. In fact, in a concurrent scenario a specification such as "go back n steps" (typical of sequential reversible debuggers such as in [3]) is not meaningful, since there is no clear understanding of which the last n actions have been. Irreversible actions and explicit rollback are dual, one specifying when it is *forbidden* to go back, and the other one specifying when it is *required* to go back. Their combination is not trivial, however, since one has to decide what to do in case of conflicts, e.g. if a rollback requires to go back past an irreversible action. We will outline a possible solution to this problem when discussing compensations.

External Control: This approach follows the separation of concerns principle: a process is potentially able to go both backward and forward, while another process is in charge of controlling it by deciding when it has to go backward

and when it has to go forward. Such an approach is suitable, e.g., for hierarchic component-based systems, where the father component may decide when and how to rollback its child, and the children notify the father in case of errors. Such a hierarchical structure for failure handling is typically the one advocated for Erlang systems [1]. External control naturally emerges in a reversible debugger: the user, through the debugger interface, decides whether the program under debugging should execute forward or should get back to a previous computation. Following the causally consistent approach, when going backward the user should specify which past action to undo, and the system should be in charge of finding its dependencies and undoing their execution. This is in contrast, e.g., to [7] where the user has to decide which actions to undo and in which order. External control has been applied also to biological reversible systems in [20]. There a reversible CCS process P is controlled by a controller process C, which is again a CCS process. The controller C always computes forward, and it constrains the possible actions of P, thus decreasing the non-determinism due to reversibility and to concurrency. This allows, together with a generalized form of prefix, to model different forms of reversibility, including reversibility that is not (always) causally consistent.

Semantic Control: In this approach the semantics of the language is extended with guidelines on whether to go forward or to go backward. Consider the following scenario: a reversible program is used to perform a state-space exploration looking for some solution of a given problem. In this case reversibility is needed to backtrack in case a branch with no solution has been taken. One can imagine to add to the history information about whether and how many times a particular path has been taken and favor paths (and directions of execution) leading to less explored areas. For instance one can label each action with the number of times it has been tried, and choose among the enabled actions (both forward and backward) one which has minimal value. It is clear that in such a way a finite state space is completely explored, allowing to find a solution if at least one exists. Another approach has recently been explored in [2], where computing steps are taken subject to some probability, and the rate of forward and backward computing steps are derived from a set of formal energy parameters. The contribution of the paper is to show that there exists a lower bound on energy costs to guarantee that a process commits a forward computation in finite average time.

4 Reversibility and Compensations

Using some forms of internal or external control, reversibility may lead to divergence. In particular, the process itself is not aware of the fact that a specific computation has already been executed, has failed, and has been rollbacked. Thus the same computation could be performed again and again, possibly forever. To avoid such a problem we put forward a solution based on *compensations*. Compensations have been proposed as a main building block for long running

transactions, first in the area of database theory [13] and then in service-oriented computing [5,4,18,14]. A long running transaction is a computation that either succeeds, or, in case of failure, it is compensated. A compensation is an ad hoc piece of code which is in charge of leading the system back to a consistent state, possibly different from the ones the execution went through. Compensations seem antithetic to reversibility, since their aim is exactly to deal with situations where rollback is not possible or not desired. However, the two concepts can be fruitfully combined. Consider any form of controlled reversibility, e.g., one based on internal control. Assume that for some of the statements of the program a compensation is defined. One can consider for instance a statement of the form A `comp` B. The idea is that during forward execution A `comp` B behaves as A. However, if its execution has to be rollbacked (possibly as part of a larger rollback), the effect of A is annulled (that is A is actually rollbacked) and then the restored A is replaced by B. Both the steps are important: rollback is needed to undo some nasty effects of A on the state, while replacing A with B is needed to avoid re-doing a try that already failed, and would probably fail again. Note that the first aspect is completely missing from the compensation approaches in the literature [5,4,18,14].

Consider a typical web service scenario. A is an invocation of a flight reservation service of some airline. Possibilities for B are for instance to execute the same booking using another airline, or to update the database of preferred airlines by adding the information that the invocation of A has failed. These two possibilities are representatives of two classes of compensations with different features. We call compensations in the first class *replacing compensations*, since they aim at doing what action A was supposed to do, but in a different way. We call compensations in the second class *tracing compensations*, since they give up on what action A was trying to do, but they just aim at keeping trace of the failure. This information will be used later on by the application. In the example one may imagine that the application uses trust as a criterion to choose the airline to be invoked, and the tracing compensation decreases the trust value of the airline whose invocation failed. In the long running transactions field instead, the main aim of compensations is to remedy the nasty effects of A, e.g. annulling the previous booking to avoid to pay for it. In our case this is done automatically by the reversibility mechanism. We call this last form of compensation *repairing compensation*.

The replacing compensation example, trying again the booking with a different airline, makes it intuitive that compensations may have their own compensations, recursively. A less evident issue is the following: as we have seen, the reversibility machinery tracks causal dependencies, thus one has to specify how the compensation is inserted in the causality relation. There are two main possibilities, one suitable for replacing compensations, and the other for tracing compensations. For replacing compensations, compensation B should take the place of A in the causality relation, since it is an alternative to it. On the other hand, a tracing compensation B is just used to update the state with information obtained by the failed attempt, thus it is not causally related to A causes,

but it is independent. To better understand the difference let us analyze what happens if a larger part of the execution has to be annulled, e.g. since, in the airline booking scenario above, the user changed its mind and decided not to travel any more. The replacing compensation "book with another airline" has to be reverted, since it is no more meaningful. This is exactly what happens since the compensation is now causally dependent on A causes. Instead, the tracing compensation "remember that the chosen airline is not good" should be applied anyway, since it can be used for further bookings later on, and thus should not depend on A causes.

So far we considered the use of compensations to avoid repeating past errors. However compensations can also be used in a way closer to their original purpose, i.e. to deal with irreversible actions. Keep in mind that whatever the support for reversibility is, there will always be actions which cannot be undone. This is mainly related to two situations: the action may be inherently irreversible, e.g. a side effect on the real world such as printing a document, or the action is in principle reversible but it is out of the control of the considered system. An example of this last possibility is a distributed application where some of the components provide support for reversibility, while others do not. In both the cases one may attach compensations to irreversible actions: reversible actions are reverted, and irreversible actions are compensated. In this last case repairing compensations are normally needed.

We can now clarify the issue of the combination of irreversible actions and rollback: in case a rollback request is issued, and it requires to undo an irreversible action, actions are undone till the irreversible action is found, and its (repairing) compensation is executed, instead of reversing it and all the actions it depends on. Note that in this setting some actions are equipped with compensations, while others are reversed, while in the previous setting each action was both reversed and compensated.

5 Conclusion

In this paper we discussed three main issues related to defining and exploiting reversibility in a concurrent scenario. The first issue was how to define mechanisms allowing to go backward and forward in a concurrent execution. The second issue was how to control reversibility, i.e. how to specify whether to go forward or backward, and up to where. The third issue was how to avoid repeating the same errors, and how to deal with irreversible actions. The first such issue has been discussed in the literature to some extent, relying on the main concept of causally consistent reversibility. Related to the second issue, this is the first time, as far as we know, that a taxonomy of the possible approaches has been defined. Concerning the third issue, the only related work in the literature is [10], which proposes a transactional mechanism with some reversibility features. This work however comes from the opposite direction, since it starts from the problem of how to define interacting transactions.

Many other issues remain to answer our original question. In particular, the mechanisms we sketched above should be fully specified, and their expressive

power has to be assessed against proposals in the literature. Our endeavor would be fully successful only if we show that using a reversible framework allows to recover, improve and combine existing techniques for dependable concurrent systems, and possibly to define new ones. Also, practical issues must be solved to make the approach usable in a real programming language. For instance, the interplay between reversibility and language features such as the type system or modules should be considered. Also, the space and time overhead due to reversibility have to be measured and minimized. See [17] for a preliminary analysis of this issue.

References

1. Armstrong, J.: Making Reliable Distributed Systems in the Presence of Software Errors. PhD thesis, KTH, Stockholm, Sweden (2003)
2. Bacci, G., Danos, V., Kammar, O.: On the Statistical Thermodynamics of Reversible Communicating Processes. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) CALCO 2011. LNCS, vol. 6859, pp. 1–18. Springer, Heidelberg (2011)
3. Boothe, B.: Efficient Algorithms for Bidirectional Debugging. In: Proc. of PLDI 2000, pp. 299–310. ACM Press (2000)
4. Bruni, R., Melgratti, H., Montanari, U.: Theoretical Foundations for Compensations in Flow Composition Languages. In: Proc. of POPL 2005, pp. 209–220. ACM Press (2005)
5. Butler, M., Hoare, S.T., Ferreira, C.: A Trace Semantics for Long-Running Transactions. In: Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.) Communicating Sequential Processes. LNCS, vol. 3525, pp. 133–150. Springer, Heidelberg (2005)
6. Collet, R., Van Roy, P.: Failure Handling in a Network-Transparent Distributed Programming Language. In: Dony, C., Lindskov Knudsen, J., Romanovsky, A., Tripathi, A. (eds.) Exception Handling. LNCS, vol. 4119, pp. 121–140. Springer, Heidelberg (2006)
7. Cook, J.J.: Reverse Execution of Java Bytecode. Comput. J. 45(6), 608–619 (2002)
8. Danos, V., Krivine, J.: Reversible Communicating Systems. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 292–307. Springer, Heidelberg (2004)
9. Danos, V., Krivine, J.: Transactions in RCCS. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 398–412. Springer, Heidelberg (2005)
10. de Vries, E., Koutavas, V., Hennessy, M.: Communicating Transactions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 569–583. Springer, Heidelberg (2010)
11. Elnozahy, E.N., Alvisi, L., Wang, Y.M., Johnson, D.B.: A Survey of Rollback-Recovery Protocols in Message-Passing Systems. ACM Comput. Surv. 34(3) (2002)
12. Eppinger, J.L., Mummert, L.B., Spector, A.Z.: Camelot and Avalon: A Distributed Transaction Facility. Morgan Kaufmann (1991)
13. Garcia-Molina, H., Gawlick, D., Klein, J., Kleissner, K., Salem, K.: Coordinating Multi-Transaction Activities. Technical Report CS-TR-2412, University of Maryland, Dept. of Computer Science (1990)
14. Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: On the Interplay Between Fault Handling and Request-Response Service Invocations. In: Proc. of ACSD 2008, pp. 190–199. IEEE Computer Society Press (2008)

15. Lanese, I., Mezzina, C.A., Schmitt, A., Stefani, J.-B.: Controlling Reversibility in Higher-Order Pi. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 297–311. Springer, Heidelberg (2011)
16. Lanese, I., Mezzina, C.A., Stefani, J.-B.: Reversing Higher-Order Pi. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 478–493. Springer, Heidelberg (2010)
17. Lienhardt, M., Lanese, I., Mezzina, C.A., Stefani, J.-B.: A Reversible Abstract Machine and Its Space Overhead. In: Giese, H., Rosu, G. (eds.) FORTE 2012 and FMOODS 2012. LNCS, vol. 7273, pp. 1–17. Springer, Heidelberg (2012)
18. Oasis. Web Services Business Process Execution Language Version 2.0 (2007), http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
19. Phillips, I., Ulidowski, I.: Reversing Algebraic Process Calculi. J. Log. Algebr. Program. 73(1-2) (2007)
20. Phillips, I., Ulidowski, I., Yuen, S.: A Reversible Process Calculus and the Modelling of the ERK Signalling Pathway. In: Glück, R., Yokoyama, T. (eds.) RC 2012. LNCS, pp. 218–232. Springer, Heidelberg (2012)