From UML Activity Diagrams to Specification Requirements[1]

Doron Drusinsky[2]
Department of Computer Science
Naval Postgraduate School
833 Dyer Road, Monterey, CA 93943, USA
ddrusins@nps.edu

Abstract

Formal verification of system-of-systems uses computer-based techniques to assure that the behavior of a subject system of systems complies with its formal correctness specifications. Such formal specifications are often created on the basis of natural-language (NL) requirement specifications. While NL documents such as marketing requirement documents and concept-of-operation (CONOPS) documents contain NL requirements, they are almost never complete, i.e., they omit necessary NL requirements. To that end, UML analysis is an increasingly popular technique for requirement elicitation. This paper describes the process of identifying NL requirements of interest from UML analysis diagrams such as activity diagrams) and Message Sequence Diagrams.

1 Introduction

In [Br], Bruggue described UML-based process for requirement elicitation. One of the first steps of this process is UML use-case analysis, which results in use-case diagrams and use-case documents. A use-case document aligns user needs with system functionality by directly stating the user intention and system response for each step in a particular interaction.

While use-case documents are written in Natural Language (NL), the UML provides diagrammatic languages that capture behavior embedded in use-cases. Two primary such languages are Activity diagrams (AD's) – used primarily to capture workflow-like behavioral, and Message Sequence Charts (MSC's) – used primarily to capture event driven behavior.

While mostly informal in UML-1, AD's and MSC's have been empowered in UML-2 to be more rigorous and accurate. The jury is still out on the formal descriptive power of these languages; some would argue that UML-2 AD's and MSC's are as formal as it gets, while other's claim that they (especially AD's) are still not formal enough. Either way, this paper focuses on the rather informal UML-1 level AD's and MSC's. This is for several reasons:

1. These are the most commonly used forms of AD's and MSC's.
2. Our tool chain does not support computer-aided verification using AD-based or MSC-based formal specification, rendering the investment in building rigorous formal AD or MSC specifications wasteful.
3. As discussed in chapter 2, a significant learning curve and significant additional investment are required to step up from a UML-1 level AD or MSC to a rigorous and formal UML-2 level diagram.

Formal requirement specifications are specifications that are readable and executable by a computer based verification system. Many formal specification languages have been described in the literature, including linear-time temporal logic (LTL) [LTL], branching-time temporal logics [CTL], and statechart assertions [Dr1]. The ultimate purpose of formal specifications is for subsequent computer-aided verification, using techniques such a model-checking, theorem proving, and run-time monitoring; [DMS] provides a three dimensional comparison of these three primary verification techniques.

Harel statecharts [Ha], currently part of the UML standard, are typically used for design analysis and implemen-

[1] The research reported in this article was funded in part by a grant from NASA. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotations thereon.

[2] Also with Time-Rover Software Inc. www.time-rover.com

978-1-4244-2173-2/08/$25.00 ©2008 IEEE

tation. In his recent book [Dr1], the author suggested using deterministic and non-deterministic statecharts-assertions for formal requirement specification and run-time verification. This approach is currently in active use by the NASA IV&V facility.

As described in [Dr1], formal specification assertions for reactive systems are typically based on NL requirements. For example, Fig. 1 depicts the statechart assertion for the NL requirement *R1*: *whenever delayRequest then no ack is permitted for 30 seconds*. Once the formal specification assertion is constructed, the process of *validation testing* is applied to that assertion to assure that the assertion's behavior actually conforms to the expected behavior of NL requirement. In other words, validation is the process of *certifying* the assertion as a trust-worthy representative of the NL requirement.

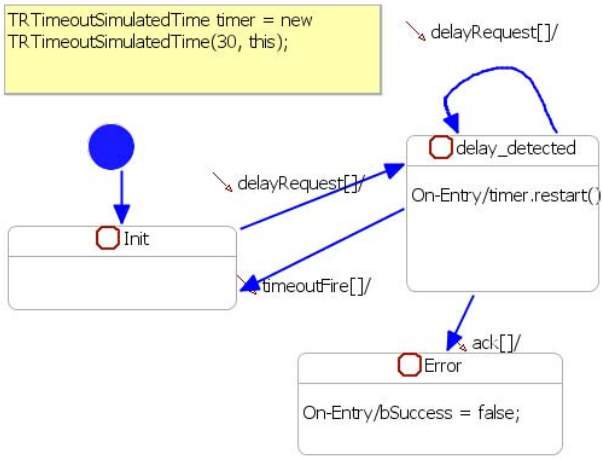

Figure 1. Statechart assertion for NL requirement R1.

Hence, the NL and formal requirement elicitation and development process described thus far is depicted in Fig. 2:

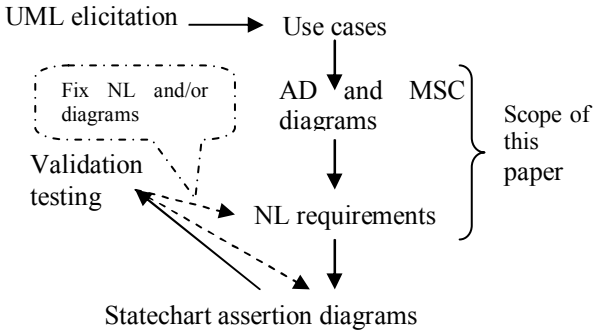

Figure 2. From UML elicitation to formal-specification assertion

This paper addresses the process of identifying *behavioral concerns* for a reactive system of systems, i.e., concerns that lead to eventual NL requirements. It does so by addressing areas of concerns from an activity diagram point of view, as illustrated in Fig. 2.

Although the paper focuses on UML AD analysis the same process can be applied to UML MSC diagrams.

2 The Formality of Activity Diagrams and MSC Diagrams

UML-1 AD and MSC diagrams are used almost exclusively as non-executable artifacts, mainly for purposes of documentation, human based analysis, and the exchange of ideas between stake holders. I coin this type of use as *wall-paper* because these diagrams typically end-up covering walls in conference rooms. In contrast, assertions are executable, and are constructed with the ultimate purpose of being used for subsequent computer-aided verification.

In their commonly used form, AD's and MSC's are informal, i.e., while they do convey a general "look and feel" for a scenario of interest, they are ambiguous when it comes to precise details of legal and illegal variations of this scenario. For example, consider the AD of Fig. 3. It does not clearly specify whether, upon fulfillment of the preconditions for this AD (*scheduled downlink*), the final activity (*Receive downlink*) *must* be executed or not (i.e., perhaps all the AD specified is that the *Receive downlink* activity *can-be* executed). Specifically, the actual system might execute activity *A*, then *B*, then *C* and *E*, but not *Receive-downlink*; the AD of Fig. 3 does not specify whether this is legal or illegal behavior.

In fact, that the AD of Fig. 3 (excluding the extension) specifies explicitly only the following two legal scenarios:

Seq-1: *schedule-downlink* followed-by activity *A*, followed-by activity *B*, followed-by parallel activities *C* and *E*, and finally activity *Receive-downlink*.

Seq-2: *schedule-downlink* followed-by activity *B*, followed-by parallel activities *C* and *E*, and finally activity *Receive-downlink*

Hence, the AD of Fig. 3 *does not* specify whether scenarios that are not explicitly depicted are legal or not, such as:

Seq-3: identical to Seq-1 but without activity B.

Seq-4: identical to Seq-1 but without activity C.

Etc.

There are two possible ways of addressing this limitation:

1. To create powerful AD-based and MSC-based formal specification languages. The following attempts at doing so have been reported in the literature:
    a. Live Sequence Diagrams (LSC's) [LSC], an extension of MSC diagrams with *must* (denoted

as *hot*, or *red*) activities, and *can* (denoted as *cold*, or *blue*) activities.

b. MSC-assertions [DS1, DST], a super positioning of statechart-assertions on top of MSC diagrams.

c. UML-2 AD and MSC diagrams [UML2]. As discussed in the introduction UML 2 has improved the power of these formalisms to a degree that they no longer specify only a small set of allowable scenarios.

2. The approach described in this paper: to create executable assertions for key concerns, to then formalize them as statechart assertions, to validate those assertions against expected behavior, and then deploy those assertions for verification.

This paper uses the second approach for the following reasons:

- With the exception of MSC-assertions, assertions created with the approaches listed in item 1 do not have associated computer-aided verification tools and cannot be used for subsequent verification.
- Our experience with MSC-Assertions (e.g., see [DST]) and LSC's has shown that as the level of formalism in the underlying diagram type increases (e.g., from informal UML-1 AD to fully specified and unambiguous UML-2 diagrams) they become considerably more difficult to use when compared with the simple, familiar, and intuitive statechart-assertion based approach.
- This paper described results of an on-going effort with NASA and was influenced to a large degree by the availability of stable and productive academic and commercial tools for validation and verification. Consequently, the most suitable formal specification tool was found to be one that supports statechart-assertions [TRSI].

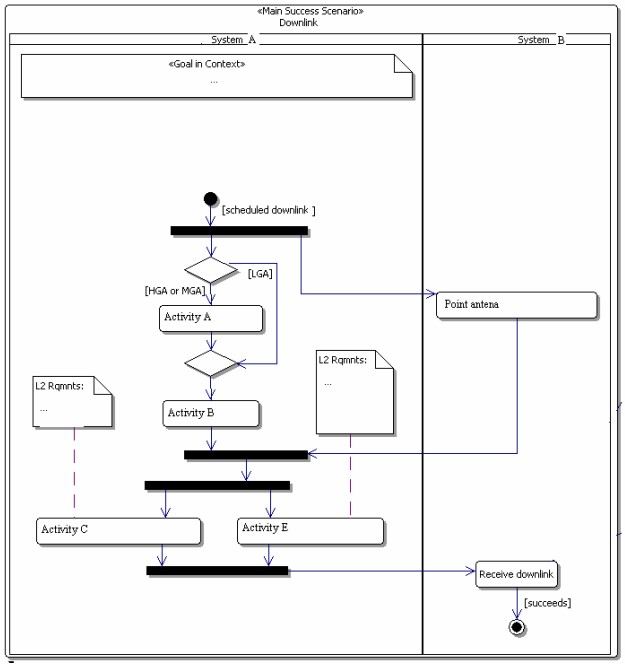

Figure 3. An activity Diagram

3 From AD to NL Requirements

3.1 *Must* vs. *Can*

As discussed earlier, the AD of Fig. 3 lists only two explicit legal scenarios. It is unlikely however that the actual system has only two possible use-case scenarios. It is therefore likely that other legal scenarios exists, some captured by some other AD's. This explains how a sequence such as Seq-3 or Seq-4 could actually happen, namely, when a *different AD* specifies such a scenario. In this case, a concern for the developer of the AD of Fig. 3 is whether Seq-3 and Seq-4 are legal or not. If illegal, then these concerns are candidates for a NL requirements, such as:

R2: whenever *schedule-downlink* then activity *Receive-downlink* must eventually be performed.

R3: whenever Activity-*B* is executed then both activity *C* and *E* must follow.

3.2 Repetitions

Consider the AD of Fig. 3 again. Say the system performs one of the the following sequences:

Seq-5: *schedule-downlink* followed-by activity *A*, followed-by activity *A* again, followed-by activity *B*, followed-by parallel activities *C* and *E*, and finally activity *Receive-downlink*.

Seq-6: *schedule-downlink* followed-by activity *A*, followed-by activity *B*, followed-by activity *A* again, followed-by parallel activities *C* and *E*, and finally activity *Receive-downlink*.

The concern raised by these scenarios is whether multiple executions of an activity violate the intention of the AD or not. A possible NL requirement that covers such a concern is:

R4: whenever activity *A* executes it should never execute again until *Receive downlink* activity executes.

3.3 Loops

Consider the AD of Fig. 4, which contains the AD of Fig. 3 with an extension for the case where downlink (the final activity) fails.

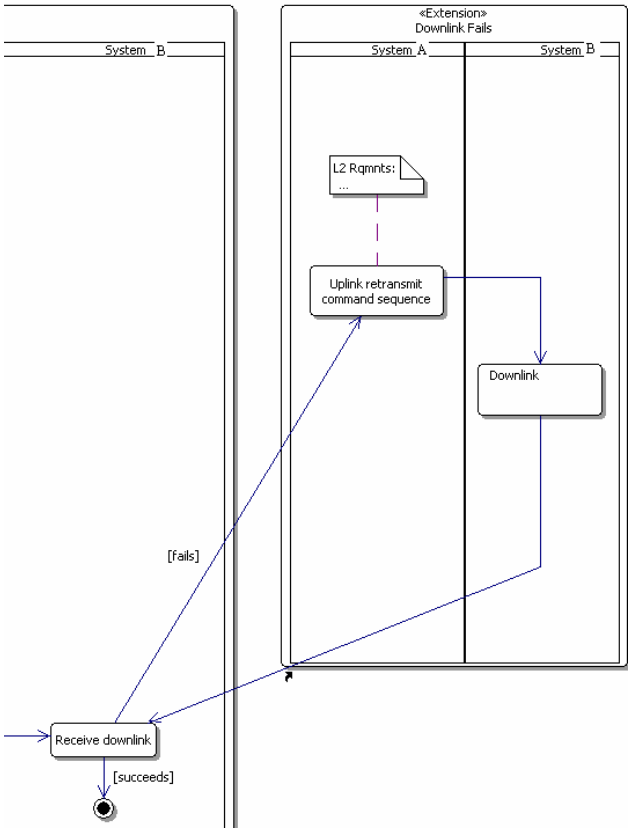

Figure 4. The AD of Fig. 3 with an extension.

An obvious concern here is how many times it is legal for the downlink to repeatedly fail (and thereby launch the extension loop to repeatedly execute).

3.4 Multi-object Systems

Suppose a plurality of objects or processes have the capacity of performing activities *A*, *B*, and *C* in the AD of Fig. 3. In such a case, a concern is whether Seq-1 is legal even if activities within Seq-1 are not all executed by the same object (e.g., object #1 executes A while object #2 executes B). Note that a swim-lane in an AD (e.g., *System-A* in Fig. 3) is commonly considered as an object type, but it's unclear from this specific AD description whether the system-of-systems is guaranteed to contain precisely one sub-system of such a type.

3.5 Reentrancy

Consider the AD of Fig. 5. Suppose that a legal scenario executes activities *C*, *D*, and *E* concurrently, as specified. Suppose also that the instrument malfunctions causing the extension to execute as specified. One immediate concern is what should happen with activities *D* and *E*? If *D* is permitted to continue then it might reach activity *F*, meanwhile, when the extension completes it will re-enter activities *C*, *D*, and *E*. This behavior could lead to activities *C* and *F* executing concurrently, a behavior that is not expected from Fig. 5, where *C* and *F* are explicitly depicted as mutually exclusive. Hence, a candidate requirement for this concern is one that forbids D and E from executing when an instrument malfunctions.

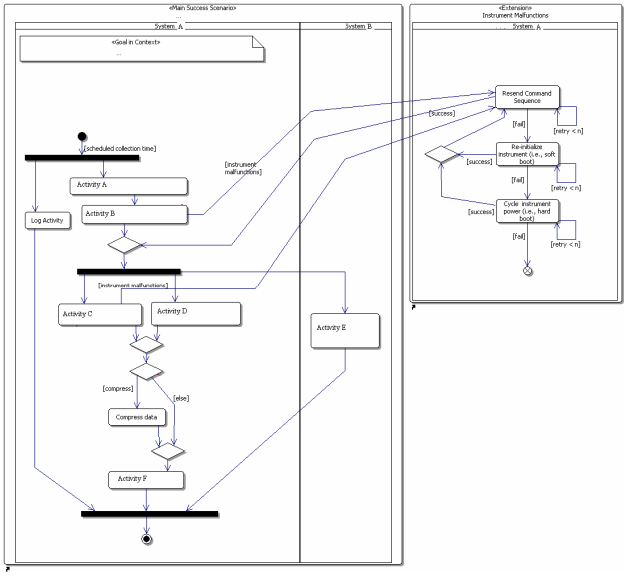

Figure 5. AD used for reentrancy discussion

Another concern that arises as a result of potential reentrancy in the AD of Fig. 5 is the following. Suppose that, as discussed above, the instrument malfunctions while activity C is executing and the resulting extension loop consumes more time than initially expected. Consequently, the concern is that during this time the AD's precondition might be satisfied again thereby inducing a replay of the scenario described in the AD; consequently activity C (and others) is executed more than once. This concern can be resolved using various kinds of NL requirements many ways, such as:

R4: System *A* instrument malfunction recovery loop must take no longer than time T (calculated based on the interval between successive collection times). Or,

R5: The AD must have a precondition that prohibits reentrance.

Note that while R5 is a straight-forward requirement to write in NL, it is can be rather costly to implement.

Reentrancy is also a potential cause for concern regarding the meaning of a successful or failed termination of the depicted scenario. For example, consider the same scenario described for the AD of Fig. 5, namely, the AD is reentered while the instrument malfunction extension is sub-scenario is playing. Afterwards, one of the two computations playing on the AD reaches the terminal state. A potential concern is that the user considers this as an indication that the scenario has completed successfully while it still has a playing computation.

Likewise, if one of the two computations terminates successfully while the other reaches a failed terminal state (at the bottom end of the extension diagram) then it is unclear whether the AD as a whole should declare success or not.

3.6 Real-time Constraints

Many of the above-mentioned derived requirements discuss eventualities that "must" occur. Unbounded eventualities are almost never testable or enforceable. Therefore, requirements such as R2 and R3 should be accompanied by a real-time constraint for the eventuality, such as:

R2': whenever *schedule-downlink* then activity *Receive-downlink* must be performed within 10 seconds.

Likewise, constraints on looping discussed in section 3.3 can be specified in terms of real-time rather than number of iterations, such as:

R6: whenever *Receive-downlink* fails it must succeed within 10 seconds.

4 Conclusion

We have shown a process for analyzing informal UML-1 activity diagrams (a process that applies to message sequence diagrams as well) that identifies potential concerns and associated natural-language requirement specifications. The extension of this process, discussed in [Dr1], enables computer aided validation and verification of the systems behavior with respect to these concerns.

References

[Br] B. Bruegge, Object Oriented Software Engineering, Prentice Hall 2003.

[CTL] E. M. Clarke and E. A. Emerson, "Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic," Proc. Workshop on Logic of Programs, D. Kozen, ed., LNCS 131, Springer-Verlag, 1981, pp. 52-71.

[Dr1] D. Drusinsky, *Modeling and Verification Using UML Statecharts*, Elsevier Publishing, 2006.

[DMS] D. Drusinsky, B. Michael, M. Shing, The Three Dimensions of Formal Validation and Verification of Reactive System Behaviors, NPS technical report NPS-CS-07-008. Submitted for publication.

[DS1] D. Drusinsky, M. Shing, Verifying Distributed Protocols using MSC-Assertions, Run-time Monitoring, and Automatic Test Generation MSC-assertions.

[DST] D. Drusinsky, M. Shing, T. Cook, Specification, Validation and Run-time Monitoring of SOA Based System-of-Systems Temporal Behaviors, Proceedings of the 2007 IEEE International Conference on System of Systems Engineering, San Antonio, TX, pp. 16-18.

[Ha] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming* 8, 1987, 231-274.

[LSC] D. Harel and R Morelly, *Come, Let's Play: Scenario-based Programming Using LSCs and the Play-Engine*, Springer, 2003.

[TL] H. Kugler, D. Harel, A. Pnueli, Y. Lu, and Y. Bontemps, "Temporal Logic for Scenario-Based Specifications", in N. Halbwachs and L. D. Zuck, editors, *TACAS*, vol. 3440 of LNCS, Springer, 2005, 445–460.

[TRSI] The StateRover site. http://www.time-rover.com

[UML2] H.K. Eriksson, M Penker, B. Lyons, D. Fado, UML 2 Toolkit, OMG Press.