Milestones: Mythical Signals in UML to Analyze and Monitor Progress

Richard Torbjørn Sanders[1] and Øystein Haugen[1,2]

[1] SINTEF, N-7465 Trondheim, Norway
richard.sanders@sintef.no
[2] University of Oslo, Dept. of Informatics, N-0316 Oslo, Norway
oystein.haugen@sintef.no

Abstract. Many applications are evolving towards Service Oriented Architecture (SOA) with technologies such as Web services. Services can be modeled platform independently through UML2 collaborations in the upcoming UML profile for services, *SoaML*. We observe an increasing need for validation of services. However, such validation is often based on syntactic descriptions of the services and of their interfaces, which are insufficient to ensure that desired liveness properties are satisfied. In this paper, we present a language construct called "milestone" embedded in UML and define its semantics using mythical signals. We show how this interpretation of milestones can be used for liveness analysis and for runtime monitoring of services. The approach is illustrated with a simple bidding service.

1 Introduction

In recent years the software community has shown large interest in adopting Service Oriented Architectures (SOA) to overcome the challenges of distributed computing [1]. SOA is an architectural approach for constructing complex software-intensive systems from a set of interconnected and interdependent building blocks. A service is a stand-alone unit of functionality available through a formally defined interface.

While SOA in itself is not tied to any particular technology, most practitioners consider contemporary SOA to be that offered by web services. Semantic web services seek to characterize what a service can provide by offering means of expressing interfaces using Web Services Description Language (WSDL) [2]. Although WSDL aims at providing a formal definition of the interface to a service, it is restricted to a static description of operations and associated messages. This may change with the upcoming response to the OMG's RFP [3]. Called SoaML [4], the UML profile for services will allow one to formally define the behavior of a service on an interface, without binding the implementation to a particular technology. SoaML prescribes modeling services using UML2 Collaborations, see Fig. 1, as we argued in [5].

We have suggested the concept of *milestones* to express the desired behavior of a service [6]. In this article we show how the semantics of milestones can be defined by *mythical signals*, and how these are modeled by specialized UML *Comments* in SoaML. Mythical signals are signals which are useful for analysis and monitoring, but can be omitted in implemented systems. This work is a result of the SIMS project [7].

The structure of this paper is as follows: first we present the rationale for milestones by way of a bidding example. Then we show how milestones are defined in UML, and how they contribute to the analysis and monitoring of progress. We also discuss related work, and finally conclude.

2 Buyers and Sellers – Progress of a Bidding Process

In this Section we introduce our illustrative example about buyers and sellers involved in a bidding process. We give an intuitive explanation of the example and then show how milestones can improve the understanding of the scenario as well as be a formal basis for liveness analysis.

Our situation is one where a seller offers an item to the market. We assume that the item is of considerable value such that a bidding process will be applied. The context is given by the UML collaboration shown in Fig. 1.

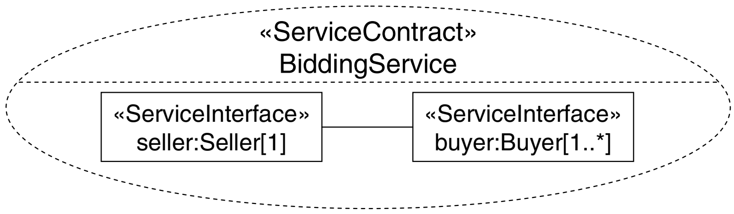

Fig. 1. Bidding service modeled as a collaboration

The seller will advertise the item for sale. We model this by assuming a message broadcast to a set of potential buyers. Some of the buyers will react favorably and return a message to indicate their interest. More information is then provided by the seller to all the buyers that have shown interest. After this preamble the bidding will start and a subset of the interested buyers will present a bid to the seller. We assume that the bid will contain additional information such as the price they are willing to pay, financing method etc. These additional pieces of information are not of much interest to our analysis and we have left them out of our simple model.

We then assume a series of bidding rounds where the seller will multicast to the remaining bidders the highest bid in the most recent round. Then the bidders may renew their bid with changed parameters. This procedure will go on for some time: it is not important for our analysis how many rounds or for how long the bidding process takes place.

Finally the bidding will terminate when the seller has selected a winner; the chosen one gets a message to pay and in return gets a contract for the item.

The bidding process is shown in Fig. 2. We have applied an augmented sequence diagram notation based on the notation and definition given in [8]. Compared to standard UML 2 sequence diagrams our notation has the capability to express broadcasting/multicasting in a precise yet compact way. The clue is that the buyer lifeline represents the whole set of buyers, but for each message (or combined fragment) we describe clearly what subset of the buyers will send or receive the message. Subsets of properties are defined in standard UML 2 [9].

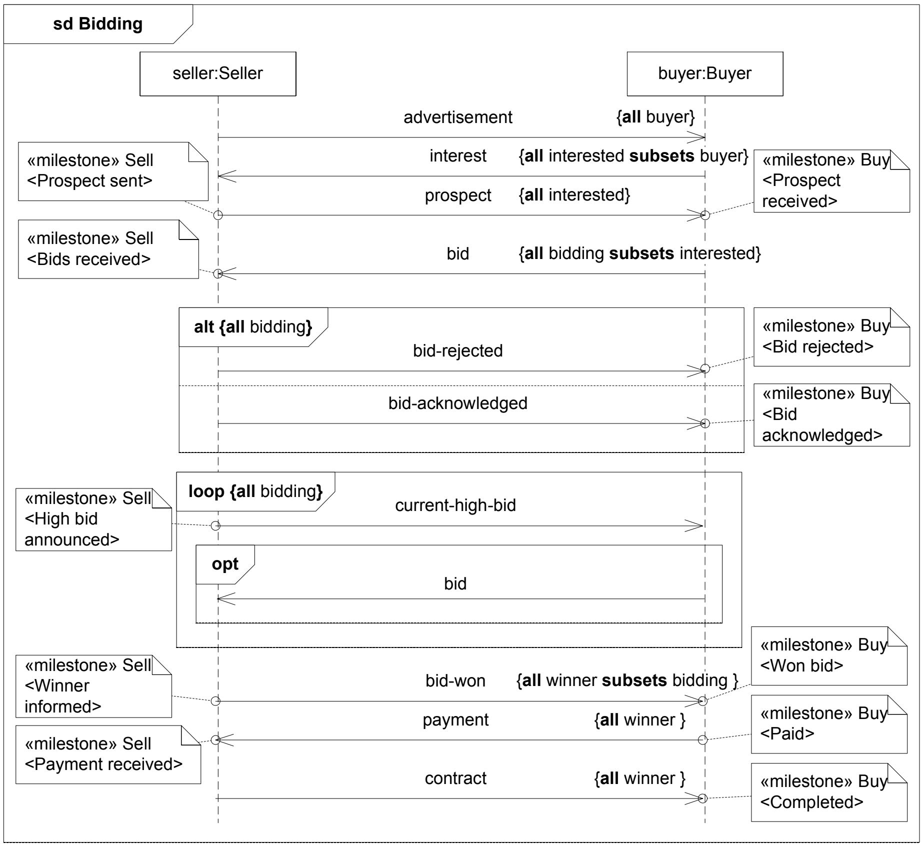
sd Bidding
seller:Seller
buyer:Buyer
advertisement {all buyer}
interest {all interested subsets buyer}
prospect {all interested}
bid {all bidding subsets interested}
«milestone» Sell <Prospect sent>
«milestone» Buy <Prospect received>
«milestone» Sell <Bids received>
alt {all bidding}
bid-rejected
bid-acknowledged
«milestone» Buy <Bid rejected>
«milestone» Buy <Bid acknowledged>
loop {all bidding}
current-high-bid
«milestone» Sell <High bid announced>
opt
bid
«milestone» Buy <Won bid>
«milestone» Sell <Winner informed>
bid-won {all winner subsets bidding }
payment {all winner }
«milestone» Buy <Paid>
«milestone» Sell <Payment received>
contract {all winner }
«milestone» Buy <Completed>

Fig. 2. The bidding process modeled by a sequence diagram

Furthermore, we have added our notation for milestones that we shall introduce shortly. Milestones express that something useful has been achieved at this point in the behavior. For instance when a seller outputs “bid-won”, the bidding process has progressed to “Winner informed” in Fig. 2.

In a multi-stage interaction, like the bidding process, it makes sense to indicate a series of partial goals, each corresponding to something worthwhile having been achieved. In actual bidding interactions, many potential buyers never get past receiving prospects or having their bids rejected; these are nonetheless identifiable partial goals and represent fully acceptable outcomes of the interaction.

Likewise, it is useful to define goals for the participants of the interaction, in this case seller and buyer. The milestones are annotated such that the collaboration role to which they refer is made clear, i.e. `Sell` and `Buy`; as we shall shortly see, `Sell` and `Buy` are in fact progress signals. Fig. 2 states that the ultimate goal of the seller is to receive payment, while for the buyer it is to receive the contract – leading up to this are the steps or sub-goals of the bidding process needed to reach these final goals.

In addition to improving the reader's understanding, milestones are useful for stating requirements. For instance, in a bidding process buyers do not want prospects from sellers that withhold their acknowledgements until the buyers have lost interest. Nor do sellers want to reward buyers with a track record of withholding payment.

Most importantly, milestones act as a formal basis for liveness analysis. Liveness analysis is concerned with systems doing something good, and milestones can be used for expressing what is considered useful. As we shall see, with milestones we can analyze at design time how objects are capable of behaving, and/or monitor at runtime how objects actually behave. This can help us ascertain whether buyers and sellers are well-behaved and follow the intensions of a service specification. For instance, in a bidding process we do not want sellers that invariably reject all bids.

One benefit of milestones is that goal achievement is easier for people to recognize and follow, saving one from time-consuming analysis of programming code, procedure calls, message exchanges and other implementation artifacts. A benefit of this approach is that we do not need additional validation models unlike what is associated with formal methods; including milestones in a design provides analysis and monitoring opportunities without increasing the complexity of the model. At runtime, monitoring progress signals is a more practical instrument than monitoring all message exchanges and performing a progress analysis on these.

2.1 Defining Milestones in a UML Context

Milestones are marks of progress placed on behavioral elements of the UML specification. In our example given in Fig. 2 we have placed the milestones on message transmissions and message receptions. We may place milestones on any behavioral element where it is well defined when that behavioral element is executed at runtime.

The milestones in Fig. 2 are depicted as comments and the way they are written may lead people to believe that they represent pure constraints, but this would be a misconception. A constraint is something that is either true or false whenever the execution reaches this element. A milestone is something that states the fact that this behavioral element has been reached. While constraints are declarative and passive, milestones are imperative and active.

On the other hand, milestones share with constraints the fact that they are not necessary for the specification to execute properly. Just as all constraints can be removed from an executable model, so can all milestones. Both constraints and milestones are descriptions that are used for analysis alone. By analysis we mean not only the formal analysis provided by automatic means, but also informal analysis done by designers.

That milestones can be removed without changing the executable definition does not mean that milestones are useless or unimportant. In fact, the same can be said about other model elements; for instance sequence diagrams are normally considered redundant relative to the executable model. There are numerous algorithms that partially or totally generate executable models from sequence diagrams, but given a UML system defined with both state machines and sequence diagrams, the sequence diagrams will be used as advanced requirements on the executions and not the source of execution themselves.

Milestones are part of this tradition. They are also the first imperative constructs suggested in a UML context that have analysis as sole purpose. What should then happen when a milestone is encountered during execution? It is not sufficient to raise a flag since the same milestone may be encountered a number of times during an execution, and only raising a flag would not distinguish between encountering the milestone once and encountering it multiple times. The numbers or frequencies of these encounters may be of significance to what we call progress.

Thus, we decide that encountering a milestone should result in sending a signal to an observer totally outside our system. The signal name is given in the milestone along with an optional ordinal number representing the degree of progress. In our example in Fig. 2 we have used one progress signal `Sell` for the seller's progress and another progress signal `Buy` for the buyers' progresses.

A formal semantics for milestones would have to enhance the formal semantics of UML as such. The enhancement would have to comprise the external observer and a precise definition of exactly when during the execution of a behavioral primitive the progress signal should be sent. A formal semantics goes beyond this paper; here we explain in UML terms how the execution of milestones is.

The progress signals are declared as any other signal, and may in fact be signals that are used for other purposes in the specification. This means that the signals may have attributes and these attributes will get the runtime values at the time of the sending of the signal; the scope of the signal arguments follows normal UML scope rules.

The signals sent when milestones are encountered are sent only for the purpose of analysis and we imagine these signals are sent to a possibly fictitious observer outside our system. Since these signals could be omitted and since they may be understood as only being present for those that analyze, we call them "mythical signals". The term has some merit, as the term "mythical variable" was coined already in the seventies [10, 11]. The term "mythical variable" was used for variables that were not needed for the execution itself, but were auxiliary variables used to facilitate the reasoning. Typically the mythical variables have represented a history of states [12, 13] and as such they are similar to our mythical signals since the sequence of these signals represents a way to trace the history of the execution. In fact, we could apply a mythical variable to represent the sequence of mythical signals.

We have contributed the concept of milestones to the upcoming standard on service modeling (SoaML) [4], where the piece of the metamodel for milestones is depicted as in Fig. 3.

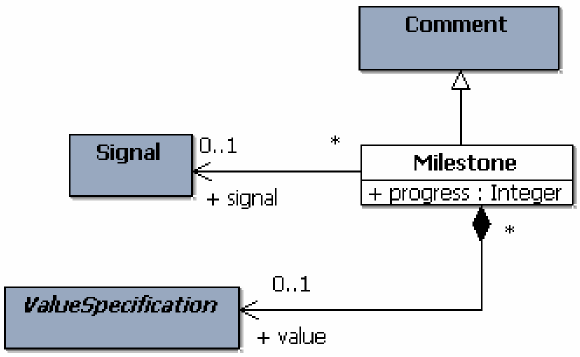

Fig. 3. Metamodel for Milestones

The metamodel expresses exactly what we have presented in this Section. A milestone is a kind of comment associated with a signal and an expression for the arguments of that associated signal. Furthermore, there is the progress value representing the degree of progress as an ordinal integer.

Milestones are not redundant in the sense that no other UML construct covers the same purpose. One may argue that sending a signal at selected places in the behavioral description can be done with ordinary UML means. This is true, but just including a number of signal-sending constructs does not serve the same purpose for the following reasons:

1. Milestones provide a uniform concept and notation across the different behavioral views of UML. Just sending signals will require different kinds of constructs for each of the behavioral diagrams.
2. Milestones are easily distinguished from sending signals that are necessary for the functioning of the system itself. This is what constitutes the mythical property of the milestones, that they are only used for analysis and not for specifying the functionality itself.
3. Milestones define sending of signals to an imaginary observer that is external to the outermost running system. Within UML there are constraints associated with sending of signals to indicate where the signal is sent. E.g. in sequence diagrams signal sending is represented by messages and these may go to the frame border. This is, however, the definition of a gate and must be matched where that Interaction is referenced. For our analytical purposes such constraints are counterproductive while they are practical and useful for pure functional purposes.

Milestones are not made superfluous by advanced model-driven debuggers either. It is possible to configure model execution support tools to report on reaching behavioral elements, but this is not well integrated with the modeling itself and it is an activity related to monitoring rather than analysis.

2.2 Progress Analysis

Validation at design time can be performed to ensure that components involved in a service will be able to interact safely with each other. We consider that components interact safely when their interactions do not lead to any unspecified signal receptions, deadlocks or improper termination.

The desired interface behavior of a participant in a service can be specified in a state machine like the one in Fig. 4 below. Here we see the specification of the interface behavior of the `Buyer` referred to in Fig. 1 and Fig. 2, where milestones are inserted at appropriate points. Interface behavior specifies the input and output signals on an interface, and is thus only a partial state machine; in particular it does not define causality of signal output. Fig. 4 constitutes what we have called a *semantic interface* [14]; it is indeed the milestones that contribute with the semantics of the interface.

Given a requirement specification detailing the interaction behavior, a component does not necessarily have to implement the complete behavior to be considered being compatible with the specification in terms of safety properties. Simply stated, a component can provide less output and accept more input than a specification, and (within

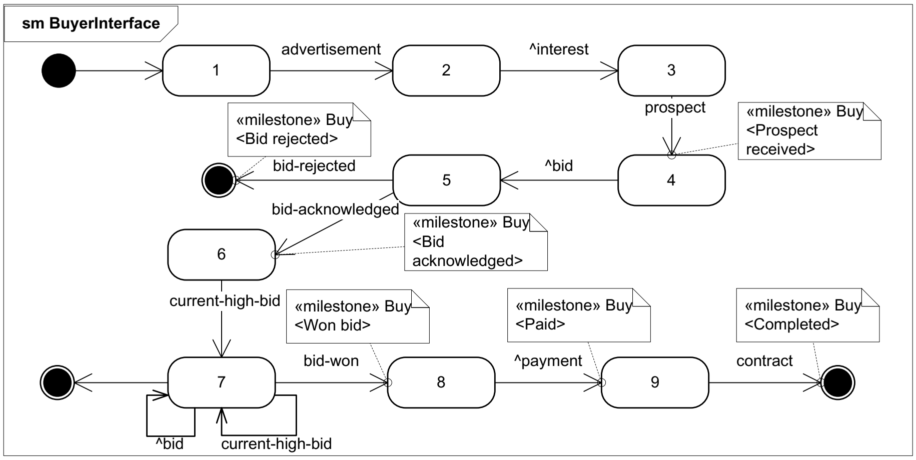

Fig. 4. Interface behavior of Buyer with milestones

certain constraints) still be safe, as we have discussed [14]. Interactions are considered safe if no unexpected signals are received and deadlocks do not arise (meaning that the peers wait endlessly for signals from each other). A safe behavior with less output is what we call a *safe subtype* [15].

However, that a component can interact safely in a service does not mean that it is useful. This is exemplified by the following diagram: Fig. 5 shows the possible behavior of a buyer that always responds with interest when it receives advertisements, but never does anything with the prospect it subsequently receives. The bidding world is full of would-be buyers that demonstrate behavior like this.

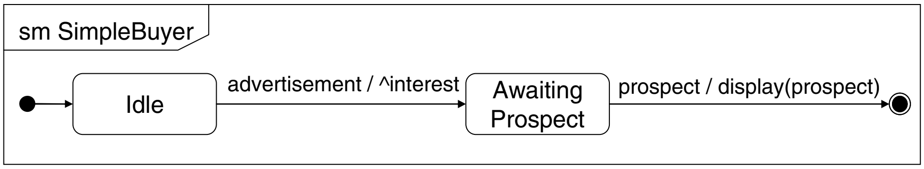

Fig. 5. Buyer behavior that is safe but not very useful

At first glance, Fig. 4 and Fig. 5 seem quite different; state names are different, and there are less states and signals in the latter. Some of these differences are due to the fact that the latter is the state machine of an object or classifier, while the former represents the interface behavior, and can be obtained by projection on an interface. Projection is due to the work of Floch [16], and is a mechanical process performed in order to simplify interface validation. In simple terms, projection removes events not visible on the interface, such as `display(prospect)` in Fig. 5, and transforms the state machine into a transition chart, e.g. the input of `advertisement` and the output of `interest` are placed in separate transitions, and auto-generated state names (numbers) are used. Projecting the state machine of `SimpleBuyer` (see Fig. 5) on the interface to the `Seller` results in the interface behavior in Fig. 6 below.

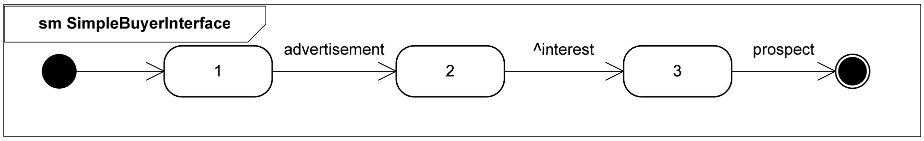

Fig. 6. Interface behavior of `SimpleBuyer`

Comparing Fig. 6 with Fig. 4 is straight-forward; we can see that `SimpleBuyer` performs the first part of the specification, ending where the specification reaches state `4`. Formally, a buyer acting according to the state machine in Fig. 5 can behave safely in the bidding service; it is capable of receiving the initial signals output from a seller[1], and does not output anything that a seller is incapable of handling according to Fig. 4. It is indeed a safe subtype, implying that it behaves safely in a bidding process. A seller would not receive any unexpected signals from such a buyer, nor would a seller wait endlessly for any signals from it, since bidding is not mandatory according to the service specification. However, seen from the perspective of a seller it is not very satisfactory, as such a buyer would never provide any bid.

This is where milestones come in. We can use milestones to analyze the behavior of an object or class and check if it is able to achieve the goals defined in a service specification. For buyers following the service behavior of the `SimpleBuyer` we see by comparing its projection in Fig. 6 with the specification in Fig. 4 that only one of the sub-goals can be achieved, `Buy <Prospect received>`, and neither the ultimate goal of seller nor buyer discussed earlier can be obtained in any interaction. Clearly, seen from the perspective of the seller, such buyer behavior is not fully satisfactory, since sellers want buyers to bid, not just to browse prospects. Analysis of the buyer behavior can disclose this; knowing this, sellers can take measures to avoid involving such participants in the bidding process.

A more serious case for the bidding process, however, would be buyers that win bids but are not able to provide payment, or sellers that never announce a winner, regardless of what bids are received. The latter case is an important one given the design of this bidding process: according to the specification, only the winner is informed, so active bidders are not able to check if a winner is ever announced. Analysis at design time can find this kind of discrepancy in the implemented behavior of a seller; monitoring at runtime, discussed below, can also reveal this.

Such design time validation exploits what is called a reachability analysis in formal methods. The example above is so trivial that no tool support is needed to perform the analysis. However, this is not the case in general; discovering errors and analyzing interaction behavior to find them can be difficult, and may require dedicated validation tools such as SPIN [17]. On the other hand, if one performs validation of collaborative behavior between components, one can simplify the analysis by focusing on interface behavior, and ensuring that the latter is well-formed (i.e. safe), meaning that nothing bad happens, and useful (i.e. live), meaning that it can achieve goals. As the example above shows, inserting milestones in interface behavior specifications can be

[1] We assume that a seller will not send messages like current-high-bid to buyers that do not submit bids. This is indeed as specified by the subset constructs in the sequence diagram.

used to validate liveness of subtypes, checking that progress can be achieved in the interactions. In the example above, the simple buyer is not able to achieve all the goals of the specification, and is thus not what we call a *live subtype* [15].

An example of more satisfactory buyer behavior is shown in Fig. 7 below. Using the validation approach mentioned above one can validate that the `SeriousBuyer` is fully goal compatible with that of `Buyer` in Fig. 4; analyzing its projection will show it to be a safe subtype, and that it contains transitions corresponding to all the milestones of the specification. This means it is a live subtype, and can achieve all the goals of the bidding process.

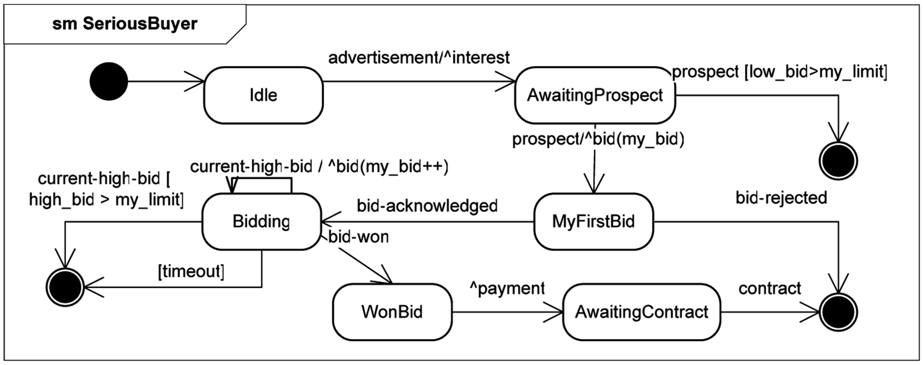

Fig. 7. A buyer that can achieve all the goals of the bidding service

The purpose of the analysis is to ascertain what progress is possible in interactions with a state machine. This does not imply that the goals are guaranteed to be fulfilled in every interaction. For instance, `SeriousBuyer` may be capable of achieving the ultimate goal of receiving a contract, but only if the bid is high enough. The analysis only shows that, given favorable conditions, this goal can be achieved. For the `SimpleBuyer`, however, the analysis concludes that no contract will ever be received.

In [15] we have suggested various kinds of milestones: graded milestones where a numeric value is specified by the label (for instance `<<Progress>> Buy (8)`), and service specific milestones of the kind used in Fig. 2 and Fig. 4. Both kinds are supported by the metamodel in Fig. 3. A graded milestone is modeled by the integer value, and can be used in making the best selection between a set of alternatives, for instance between a set of service implementations. A number of implementations may be compatible with a service specification, and a service discovery mechanism can select the implementation which exhibits the highest progress level.

Milestones are a mechanism that can be used for various needs; the analysis needs will determine what behavioral elements they are attached to. As can be seen from the metamodel in Fig. 3, milestones are specializations of comments; while comments can be attached to any model element in UML, milestones should only be attached to behavioral elements such as:

- MessageOccurrenceSpecifications in interactions (as exemplified in Fig. 2)
- Transitions in state machines (as exemplified in Fig. 4)
- ControlFlows in activities

For each of the different behavioral elements on which we may attach milestones we need to define precisely at what time instance at runtime the progress signal should be transmitted. For the three examples above, the MessageOccurrenceSpecification is not problematic as the associated event at runtime is normally considered to take zero time. The other two need more careful consideration. In most runtime situations transitions and control flows can also be considered instantaneous, but in cases where they are not we may define the progress as transmitted when the transition is finished or the control flow has given the control to the next activity node.

In the context of SoaML, service specifications seem natural candidates for exploitation of milestones; with this in place, service implementations can be validated with respect to their capabilities of fulfilling the goals expressed by the milestones.

Note that the examples presented here are simple, and do not demonstrate analysis of details such as guarded transitions. Furthermore, the analysis of interface behavior assumes that output eventually well be sent, which may not always be the case; validation of such properties using traditional state space exploration can be performed as a supplement - see [14, 15, 18] for further details of the validation approach. The benefit of milestone analysis lies primarily in the ease of use and understanding of the human designer, and the smaller size of the state space to be explored by machines.

2.3 Progress Monitoring

While we favor performing a comprehensive analysis of the models to establish progress and liveness, we also realize that most modelers do their analysis either through inspection or through testing. Formal analysis of milestones does not make testing obsolete. There may be characteristics of the system that are too difficult or too time consuming to analyze by symbolic means. Assume that there are strict time requirements on the bidding rounds, e.g. that a bidding round should not exceed one hour. A symbolic analysis of this requirement would require a lot of extra information about the behaviors of the bidders, and most certainly in a real situation the requirement could not be proved correct. Monitoring the progress on the other hand, requires only that the external observer (or in this case a "monitor") is actually implemented and the additional requirements on the progress checked by the implemented observer.

This monitoring could be compared with a special purpose debugging system or trace system. We could implement it as a state machine that consumes the mythical signals and reacts to them by compiling aggregate measures or performing checks on the fly.

In agile modeling one advocates small steps where every step is represented by an executable model. This is a very effective approach as long as it is easily established that the early immature systems perform what they should. Milestones and progress monitoring represent a lightweight approach to establishing that an immature system actually does something good without having to add all kinds of extra instrumentation to the model that must be removed later. The milestones may remain in the system, and the progress monitor may later choose not to react on certain progress signals.

3 Related Work

Clint [10] already in 1973 talks about "dummy statements" that cause "mythical" pushdown stacks to be updated with the new values of selected variables and thus recording the ongoing changes of the values. This is in fact quite similar to our approach of sending signals, only that he chose to keep the registration within the program. His aim was to prove correctness of co-routines and ours is to prove liveness of systems with concurrent, interacting processes.

In [11] Dahl applies mythical variables to count the number of times certain constructs are executed. This is again similar to our milestones as the mythical variable shows condensed information about the progress of the total program. Furthermore these mythical program variables are only meant for program analysis as their primary purpose is to appear in invariants that are used to prove the correctness of the program. [12] and [13] bring this technique one step further as the mythical variable is used to hold the whole history of the program.

The concept of milestones is inspired by mechanisms in traditional model checking, specifically the marking of so-called *progress states* in Promela [17]. While progress states markings are a mechanism used to detect non-progress cycles and livelocks in validation models, milestones are inserted into ordinary UML models in order to express, validate and monitor useful behavior, i.e. liveness in broad terms.

Milestones express the fulfillment of goals in interactions, and are a means of achieving automatic reasoning of goal achievement. The concept of goals is not unique to our work; for instance Business Motivation Model (BMM) defines the concepts of ends and goals [19]. In BMM, an *end* is something the business seeks to accomplish. An end does not include any indication of how it will be achieved. In BMM a *goal* is a statement about a state or condition of the enterprise to be brought about or sustained through appropriate means. The definitions of end and goal are not precise; the examples in [19] show normally only natural language. And although BMM goals can be formalized into OCL statements, this is less than what is desired; no algorithm can assess these goals, unlike the milestone approach we present here. Milestones seem to be a practical way of reasoning over goals.

4 Conclusion

In this article we have presented how the semantics of milestones is defined by so-called mythical signals, and how this concept can be included in extensions to UML such as the upcoming UML profile for services (SoaML). Milestones can be used to analyze and monitor service behavior, and (differently from model checking) do not require the construction of validation models; instead, milestones are embedded in ordinary UML models, to the benefit of the modeler.

We have discussed opportunities for such analysis and monitoring in terms of a simple bidding process. The application of milestones is not limited to toy examples; in ongoing research we are evaluating its use in mobile services [7].

Acknowledgements

Our work is funded by the European Community under the Sixth Framework Programme, contract FP6-IST-027610 SIMS. The work is also partially supported by the MoSiS project ip06035, a project within the ITEA 2 – Eureka framework.

References

1. Erl, T.: Service-Oriented Architecture - Concepts, Technology, and Design, 6th edn. Prentice Hall, Englewood Cliffs (2006)
2. W3C, Web Services Description Language (WSDL) Version 2.0 (2006), `http://www.w3.org/TR/2006/WD-ws-cdl-10-primer-20060619/`
3. OMG, UML Profile and Metamodel for Services (UPMS) RFP - soa/06-09-09 (2006), `http://www.omg.org/cgi-bin/doc?soa/2006-9-9`
4. OMG, Service oriented architecture Modeling Language (SoaML) - ad/2008-08-04 (2008), `http://www.omg.org/cgi-bin/doc?ad/08-08-04.pdf`
5. Sanders, R.T., et al.: Using UML 2.0 Collaborations for Compositional Service Specification. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 460–475. Springer, Heidelberg (2005)
6. Sanders, R.T., Floch, J., Bræk, R.: Dynamic Behaviour Arbitration using Role Negotiation. In: Next Generation Networks. Eunice 2003, Budapest, Hungary (2003)
7. SIMS - Semantic Interfaces for Mobile Services (2008), `http://www.ist-sims.org`
8. Haugen, Ø.: Challenges to UML 2 to describe FIPA Agent protocol. In: ATOP @ AAMOS 2008, Estoril, Portugal (2008)
9. OMG, UML 2.0 Superstructure Specification, Revised Final Adopted Specification, ptc/04-10-02, Object Management Group, Needham, MA, USA (2004)
10. Clint, M.: Program Proving: Coroutines. Acta Informatica 2, 50–63 (1973)
11. Dahl, O.-J.: An approach to Correctness Proofs of SemiCoroutines. In: Blikle, A. (ed.) MFCS 1974. LNCS, vol. 28, pp. 157–174. Springer, Heidelberg (1975)
12. Gjessing, S., Munthe-Kaas, E.: Trace Based Verification of Parallel Programs with Shared Variables. In: Twenty-Second Annual Hawaii International Conference on System Sciences, Kailua-Kona, HI, USA (1989)
13. Johnsen, E.B., Owe, O.: Object-Oriented Specification and Open Distributed Systems. In: Owe, O., Krogdahl, S., Lyche, T. (eds.) From Object-Orientation to Formal Methods. LNCS, vol. 2635. Springer, Heidelberg (2004)
14. Sanders, R.T., et al.: Service Discovery and Component Reuse with Semantic Interfaces. In: Prinz, A., Reed, R., Reed, J. (eds.) SDL 2005. LNCS, vol. 3530. Springer, Heidelberg (2005)
15. Sanders, R.T.: Collaborations, semantic interfaces and service goals: a way forward for service engineering, Norwegian University of Science and Technology (NTNU), Trondheim (2007), `http://www.diva-portal.org/ntnu/abstract.xsql?dbid=1476`
16. Floch, J.: Towards Plug-and-Play Services: Design and Validation using Roles, Norwegian University of Science and Technology (NTNU), Trondheim (2003)
17. Holzmann, G.J.: Design and Validation of Computer Protocols. Prentice Hall, Englewood Cliffs (1991)
18. SIMS deliverable D2.1 - Language and Method Guidelines, 1st version (2007), `http://www.ist-sims.org/`
19. OMG, Business Motivation Model (BMM) Specification dtc/07-08-03 (2007), `http://www.omg.org/docs/dtc/07-08-03.pdf`