Creation and Evaluation of Formal Specifications for System-of-Systems Development

Drusinsky, Doron

October 10-12, 2005
Creation and Evaluation of Formal Specifications for System-of-Systems Development

Doron Drusinsky
Computer Science Department
Naval Postgraduate School
Monterey, CA, USA
drusins@nps.edu

Man-Tak Shing
Computer Science Department
Naval Postgraduate School
Monterey, CA, USA
shing@nps.edu

Abstract - Studies have suggested that formal specifications and lightweight formal methods help improve the clarity and precision of the requirements specification. This paper describes a process to augment the current informal approaches to system-of-systems development by introducing temporal assertions to capture the safety-critical and mission-essential system requirements and runtime model checking to evaluate the system designs and implementation. The process allows users to develop and validate temporal assertions iteratively via simulation with multiple scenarios, and to use the assertions to automate the testing of the system-of-systems under development as well as armor-plating the target system against any unexpected behaviors at runtime.

Keywords: Formal specification, temporal assertion, prototyping, run-time execution monitoring.

1 Introduction

We define a system-of-systems (SoS) as a federation of existing systems and developing systems that provide an enhanced capability greater than that of any of the individual systems within the system-of-systems. The individual systems making up of a system-of-systems are often developed for a different context and subjected to a different set of constraints than those of the system-of-systems. The development of complex systems-of-systems poses many challenges [1,2]. These systems are very complex (often distributed, heterogeneous, network-centric, and software intensive) and yet have to be highly dependable. Feasible timing and safety requirements for these systems are difficult to understand, formulate and satisfy. There is a growing interest in using object-oriented analysis and design methods in conjunction with the Unified Modeling Language (UML) to develop complex systems-of-systems [3]. These methods usually start with Use Case analysis to identify user needs. Functional and non-functional requirements obtained from the Use Case analysis are often written in natural language, resulting in ambiguous and contradictory specifications whose errors are only discovered late in the development process.

Run-time Execution Monitoring of formal specification assertions (REM) is class of methods of tracking the temporal behavior, often in the form of formal specification assertions, of an underlying application. REM methods range from simple print-statement logging methods to run-time tracking of complex formal requirements (e.g., written in temporal logic) for verification purposes. NASA used REM for the verification of flight code for the Deep Impact project [4]. In [5], we showed that the use of run-time monitoring and verification of temporal assertions, in tandem with rapid prototyping, helps debug the requirements and identify errors earlier in the design process. Recently, REM has been adopted by the U.S. Ballistic Missile Defense System project as the primary verification method for the new ballistic missile battle manager because of its ability to scale, and its support for temporal assertions that include real-time and time series constraints [6].

Formal specifications and lightweight formal methods help improve the clarity and precision of the requirements specification [7]. The process of specifying requirements formally enables developers to gain a deeper understanding of the system being specified, and to uncover requirements flaws, inconsistencies, ambiguities and incompletenesses [8]. Unfortunately, users of formal methods often discover, late in the development process, that their formal requirements are incorrect, greatly reducing the effectiveness of the formal method approach.

This paper is concerned with early development of correct formal specifications. The artifacts resulted in this process can be formally analyzed to check for requirements inconsistencies, or used to derive other properties of the specified system. The formal specifications also provide the basis for test automation as well as runtime fault detection and exception handling [9].

The process described in this paper augments the current informal approaches to system-of-systems development by introducing temporal assertions to capture the safety-critical and mission-essential system requirements and runtime model checking to evaluate the system designs and implementation (Figure 1). The process starts with requirements expressed in natural language for a
specific set of scenarios. Selected safety-critical and mission-essential requirements are translated into formal temporal assertions, which are evaluated against the scenarios using the DBRover temporal logic simulator [10] or the StateRover/JUnit statechart assertion simulator [11] for correctness. Correct assertions are input to the DBRover or StateRover code generator to generate executable timing specifications, which are then used to instrument the prototype system or target system for runtime monitoring and model checking.

Figure 1. Assertion use in system-of-systems development

The rest of the paper is organized as follows. Section 2 provides an introduction to formal specifications in temporal logic and TLCharts. Section 3 presents an example that illustrates the creation and evaluation process for temporal assertions using the DBRover system. Section 4 presents a discussion on the approach and draws some conclusions.

2 Temporal Logic and TLCharts

Published REM methods typically use temporal logic as a specification language [12,13]. Temporal Logic is a special branch of modal logic that investigates the notion of time and order. This paper focuses on the use of temporal logic for specifying desired properties of discrete systems. Readers can refer to [14] for a variant of temporal logic for specifying properties of continuous systems.

Linear-time Temporal Logic (LTL) is an extension of propositional logic where, in addition to the well-known propositional logic operators, there are four future-time operators (◊Eventually, □Always, U-Until, O-Next) and four dual past-time operators. Pnueli [15] suggested using LTL for reasoning about concurrent programs. Since then, several researchers have used LTL to state and measure correctness of concurrent programs, protocols, and hardware (e.g., [16]). Metric Temporal Logic (MTL) was suggested by Chang, Pnueli, and Manna as a vehicle for the verification of real-time systems [17]. MTL extends LTL by supporting the specification of relative-time and real-time constraints. With MTL, all four LTL future-time operators can be characterized by relative-time and real-time constraints specifying the duration of the temporal operator. Temporal Logic with Time Series constraints (MTLS) was suggested by Drusinsky as an extension of MTL which enables temporal specifications to provide assertions about time-series properties such as stability, monotonicity, and min-max values [12]. For example, the following automotive cruise control code contains a stability assertion (using embedded TemporalRover syntax [10]) requiring speed to be 5% stable while cruise is set and not changed:

```java
void cruise(boolean cruiseSet, boolean cruiseChange,
        boolean cruiseOff, boolean cruiseIncr, int speed) {
    // Cruise Controller functionality
    /* TRBegin
       ... /* Cruise Controller functionality */
       TRAssert{Always ((cruiseSet) => (speed*0.95 < speed' && speed' < speed*1.05))
         // Until $speed$ {cruiseChange || cruiseOff})
     => {[...]} // user actions
    TREnd */
```

In the example speed is a temporal data variable, which is associated with the Until temporal operator. This association implies that every time the Until operator begins its evaluation, possibly in multiple instances (due to non-determinism), the speed value is sampled and preserved in the speed variable of this instance of the Until; this value is referred to as the pivot value for this Until node instance. Future speed values used by this particular evaluation of the Until statement are referred to using the prime notation, i.e., as speed'. Hence, if the speed value was 100Km/h when cruiseSet is true, then the pivot value for speed is 100, while every subsequent speed is referred to as speed' and must be within 5% of the pivot speed value.

Note how speed is declared using the $speed$ notation to be a temporal data variable associated with the Until operator. This declaration indicates to the Temporal Rover that it should be sampling a pivot value from the environment in the first cycle of the Until operators lifecycle, and to refer to all subsequent samples of speed as speed'.

Drusinsky recently suggested TLCharts as hybrid of Harel Statecharts and temporal logic [18,19]. TLCharts visually and intuitively resemble Harel Statecharts while enabling non-determinism, negation and temporal-logic conditioned transitions. This is useful for specifying abstract non-deterministic temporal properties inside a statechart specification.
Figure 2. The TLChart specification for the infusion pump controls software

For example, Figure 2 shows the TLChart specification of the following behavior of an infusion pump control software:

"Between every infusionBegin and an End-condition session, a keyPressed must be repeatedly sensed within any two-minute intervals or else an alarm must sound within 10 seconds and until keyPressed is sensed. Also, once the alarm sounds then the assertion has succeeded and no more alarms are permitted. The End-condition is defined infusionEnd being sensed until infusionBegin is sensed."

which can be formally written in MTL as:

\[
\text{Always } (\text{infusionBegin implies} \\
\quad (\text{infusionBegin } \lor \text{keyPressed}) \implies \\
\quad \quad (\text{Always } \neg \text{alarm}) \land \\
\quad \quad (\text{Next Eventually}_{\leq 120} \text{keyPressed}) \\
\quad \quad \lor (\neg \text{keyPressed Until}_{[120,130]} \text{alarm} \\
\quad \quad \quad \land \text{Until} (\text{keyPressed } \land \text{Always } \neg \text{alarm})) \\
\quad ) \\
\quad ) \land \text{Until} (\text{infusionEnd} \\
\quad \quad \land \text{Until} (\text{infusionBegin } \land \text{infusionEnd}))
\]

TLCharts provides a coherent uniform formalism for a hybrid of statecharts and temporal logic. A TLChart input string represents a sequence of combinations of stimuli and corresponding system responses; for example, a sequence may contain keyPressed - generated by the environment, as well as alarm - a system generated response. The TLChart in Figure 2 describes legal (accepted) and illegal (rejected) sequences. From a verification standpoint, a rejected string means that the systems behavior does not comply with the specification, typically due to an incorrect system reaction to the input stimuli. This application of diagrams for specification rather than programming and design explains the existence of a sink state (the Error state), which does not typically exist in a design phase statechart. TLChart’s flavor of non-determinism incorporates the specification of both good and bad computations with ambiguities resolved via a priority-based resolution scheme. Hence, a sequence of input events that ends up in both the Done state and the Error state in Figure 2 is a legal sequence because the Done state has higher priority than the Error state.

In [11] Drusinsky described the application of specification assertions to monitor and armor-plate statechart models. Using statechart assertions rather than temporal logic assertions to capture temporal assertions is most useful when the SoS, or some of its constituent subsystems, is modeled using UML statecharts.

3 Temporal Assertion Development

Typically, formal specification assertions are created from a conceptual requirement as understood by the primary modeler. This requirement is often a derivative of the main scenario resulting from the Use Case analysis. When such an assertion is deployed it is often incorrect and does not work as intended. Possible reasons for an incorrect assertions are:

1. Incorrect cognitive understanding of the requirement. This situation typically occurs when the requirement was driven from the use case’s main success scenario, with insufficient investigation of other scenarios.
2. Incorrect translation of the requirement, as understood by the modeler, to natural language.
3. Incorrect translation of the natural language specification to a formal specification.

Consequently, we propose the following iterative process for assertion development (Figure 3).
3.1 From Natural Language to MTL

Regardless of what formal notations or formal methods were used, UML modelers always start their requirements discovery process based on some scenarios involving the system and its environment, and express their understanding of the expected behavior or properties of the system informally with natural languages. For example, the modeler may come across a scenario where there is a need for the Missile Defense System Command and Control software to handle emergency requests. He expresses the requirements in English as follows:

“When a request is active, then within one minute an acknowledgement should be generated and should be contiguous active until the request subsides.”

To translate the requirements into formal assertion, the modeler first introduces one predicate for each important concept (events/responses/guarded conditions) in the above English statement:

Request = “a request is active”
Ack = “an acknowledgement is generated and active”

Next, he uses timing diagrams to analyze the temporal properties of the requirement (Figure 4).

3.2 Validating the assertion via simulated scenarios

The effectiveness of lightweight formal methods in system-of-systems development depends heavily on the modeler’s ability to understand the requirements and to express them correctly as a formal specification. Past studies [20,21] showed that incorrect requirements specifications are the primary cause of major software faults. Hence, it is crucial for engineers to validate the correctness of the formal specifications as early as possible in the system-of-systems development process. In the proposed process, engineers can test their assertions against different scenarios (expressed as timing diagrams) using the standalone DBRover temporal rule simulator without writing any simulation code.

For example, assume that the English statement in Section 3.1 is formally written as:

\[\text{Always ( Request implies Eventually} \leq 1 \text{ min Ack Until Not Request)}\]

Figure 6 shows the DBRover temporal rule simulator set up for the first scenario. In this scenario, the Request predicate is set to true after 1 minute into the simulation and remains true for the next 90 seconds, and the Ack predicate is set to true less than one minute after Request becomes true, remains true for 30 seconds, and then turns false before Request becomes false. The modeler expects the assertion to fail because Ack becomes false before Request subsides. After setting the simulation parameters, the modeler presses the Simulate button (not shown in Figure 6) and a simulation will take place, displaying the output sequence in the output zone. The output shows that the assertion results in a permanent failure at the 10\textsuperscript{th} cycle because Ack does not remain active until Request becomes
false, which matches the intent of the natural language statement.

To test the correctness of the assertion, the modeler proceeds to create three more scenarios shown in Figure 7-9.

The second scenario (Figure 7) represents a typical case where Ack remains true until Request becomes false. The third scenario (Figure 8) represents the case consisting of multiple Request. The assertion results in a permanent failure at the 18th cycle because Ack fails to become active within 1 minute after Request becomes true. The fourth scenario (Figure 9) represents the case with an unusually short Request interval. Since Request remains true for less than a minute, the condition “Eventually 1 min Ack Until Not Request” becomes true once Request subsides in the 11th cycle, even though the rule is temporarily violated, due to the absence of an Ack, in the 10th cycle. This phenomenon is due to the semantics of the temporal logic Until operator, where a Until b succeeds in the very first cycle when b becomes true. Not Request is true on cycle 11, and therefore Ack Until Not Request is true on cycle 11. The fourth scenario is clearly an example of a scenario not considered by the modeler when the requirement was conceived. It is possible that the cognitive expectation for this scenario is to see the assertion failing, as do all scenarios in which Ack subsides before Request. In this case, the formal specification needs to be adjusted accordingly. A possible new formal specification is therefore:

\[
\text{Always ( Request implies ( ( Eventually 1 min Ack ) \land ( Always 1 min Ack implies Ack Until Not Request ) ) )}
\]

Fig. 10 contains the simulation of this new assertion under the fourth scenario. Clearly, it fails on the 10th cycle.

### 4 Discussion and Conclusion

The use of formal specifications and lightweight formal methods can increase the level of dependability of the complex system-of-systems. However, the effectiveness of such approach hinges on the ability to understand the system requirements correctly and to translate them into formal specifications accurately. We have shown a process for early development and validation of formal specifications. The proposed process helps ensure the correctness of formal requirements per the modeler’s expectations, under a wide variety of scenarios. It is crucial to complete this phase early in the design process, to avoid a situation where incorrect formal specification assertions are deployed and used in later design phases, thereby slowing the overall development process.

The proposed process is made feasible with the help of the DBRover Temporal Rule Validation System. The DBRover temporal logic simulator allows modelers to set up the scenarios as simple timing diagrams. Modelers can use the DBRover GUI to specify the simulation length (counted in cycles) and the mapping of cycles to real-time (as real-time segments, where each segment has its own real-time unit and the number of such units in the segment). DBRover also supports time-series specification within the temporal rules by allowing the modelers to assign integer values to the temporal data variables. Simulation inputs and outputs for the current scenario are saved with their rules in the rule-set file, and simulation scenarios can be saved and loaded (exported and imported) to and from files using the Edit menu of the DBRover GUI.
Acknowledgement

The authors would like to thank the anonymous reviewers for their very helpful comments. The research reported in this article was funded in part by a grant from the U.S. Missile Defense Agency. 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.

References


