Well, you didn't say NOT to! A formal systems engineering approach to teaching an unruly architecture good behavior

Giammarco, Kristin

Elsevier
Abstract

This paper proposes a formal modeling approach for predicting emergent reactive system and system of systems (SoS) behaviors resulting from the interactions among subsystems and among the system and its environment. The approach emphasizes specification of component behavior and component interaction as separate concerns at the architectural level, consistent with well-accepted definitions of SoS. The Monterey Phoenix (MP) approach provides features for prediction of emergent SoS behaviors. An example highlights limitations of current modeling languages and approaches that hinder prediction of emergent behavior, and demonstrates how the application of MP can enhance SoS modeling capability through the following principles:

- Model component interactions as general rules, orthogonal to the component behavior.
- Automatically extract possible scenarios (use cases) from descriptions of system behavior.
- Test system behavior against stakeholder expectations/requirements using scenario inspection and assertion checking.

MP provides a new capability for automatically verifying system behaviors early in the lifecycle, when design flaws are most easily and inexpensively corrected. MP extends existing frameworks and allows multiple visualizations for different stakeholders, and has potential for application in multiple domains.

© 2013 The Authors. Published by Elsevier B.V.
Selection and peer-review under responsibility of Missouri University of Science and Technology

Keywords: architecture, formal methods, scenario generation, system of systems, complex systems

1. Introduction: Behavior as a fundamental aspect of design

Most of us have had the fruit of our labor bite us at one point or another. A system we proudly produce defies our design intentions and does the completely unexpected, at best embarrassing us and at worst failing our customers. It is not that we cannot anticipate the range and extent to which our system can misbehave, but our failure to describe in enough detail how the system is to behave, not only under the predictable circumstances or specific use cases we imagine, but also under the conditions to which the system is ultimately subject in its surrounding environment, the stomping grounds of emergent behavior. For some time, a more comprehensive modeling approach has been needed to predict complex adaptive system behaviors resulting from the interactions among systems.

Consider this definition of a System of Systems (SoS): “a set or arrangement of systems that results when independent and task-oriented systems are integrated into a larger systems construct, that delivers unique capabilities and functions in support of missions that cannot be achieved by individual systems alone” [16] (italics added by author). For our SoS modeling approaches to support independent system behavior models and their subsequent integration, they must address each system and the system interactions as separate concerns. Separation of concerns is a design principle adopted by the software engineering and computer science communities to write highly cohesive software modules such that each module is associated with exactly one main function, and to reduce unnecessary coupling among modules within a software program, such that a given module needs to access only a minimum of other modules to perform its functions. Just as programmers use this principle to keep their code organized and maintainable, systems and SoS engineers have use for this concept in structuring their system behavior models.

* Corresponding author. E-mail address: kngiamma@nps.edu.
This paper describes how the Monterey Phoenix (MP) system architecture modeling framework [2][3][4] can be used to structure independent models of “task-oriented” systems, and subsequently integrate them into a larger construct to enable prediction of SoS behavior. This paper uses simple authorization scenario examples to demonstrate how to treat system behavior and system interaction as separate concerns to facilitate understanding of SoS behavior.

2. Related work: Current approaches to behavior modeling

The Functional Flow Block Diagram (FFBD) notation was developed in the 1950’s to show system functions and their chronological order of execution [12]. The Enhanced FFBD (EFFBD) was developed in the 1990’s to show information flow on the diagrams as inputs/triggers and outputs [11]. The Unified Modeling Language (UML) [5] has four behavior diagrams: activity, sequence, state machine, and use case. The System Modeling Language (SysML) [13] was developed to extend UML for application on the systems scale, and directly reuses all behavior diagrams except the activity diagram, which has been modified from UML for consistency with the EFFBD and to support a continuous flow of matter or energy [7]. The notion of an xFFBD has been proposed to extend EFFBD with additional formalisms to make it more expressive [1]. Although these notations have been successfully used in modeling slices of system behavior and interaction, none are presently used to model the behavior of each component and the interaction of each component with other components in its environment, as separate concerns, nor do existing frameworks such as the DoD Architecture Framework [6] address this separation of concerns when describing event-based interactions.

Many system models describe only a subset of possible behaviors with assumptions about possible system interactions in specific scenarios or use cases, since behavior of external components may be outside the scope of the component under design. This practice, however, prevents the opportunity to observe behaviors that result from combinations of interactions that fall outside the scope of the assumptions made about external component behavior.

As an example, consider a simple User Authentication scenario done internal to a system:
1. User provides a general identification.
2. System requests unique identification.
3. User provides a unique identification.
4. If the credentials are valid, the System authorizes the User to access the services; otherwise the System notifies the User that credentials are invalid and the user may re-attempt access up to two more times.
5. The User or the System ends the session.

This narrative gives rise to at least two possible use cases: user authentication succeeds and user authentication fails. The EFFBD activity model in Fig.1. graphically depicts behavior for the scenario above, minus the access re-attempts. The approach taken in this diagram illustrates some generally accepted conventions when modeling with EFFBDs [10][11].

The EFFBD uses functional activities transforming inputs into outputs; exit conditions documenting possible outcomes of an activity; and inputs/triggers and outputs consisting of matter or energy consumed or produced by an activity. In this notation, it is typical for a component to have specific activities it performs organized on one branch of a parallel construct, similar to the use of swim lanes on a SysML activity diagram. In this example, the diagram includes a main branch for each component in the thread: the User (an “external” component) and the System (component under design). Conditions leading to different possible behaviors based on the outcome of the credential verification are included on the System branch. The User functions in this example, however, do not exhibit any structured logic for User behavior. For example, C.3 Access Services and C.4 Process Access Failure are shown in parallel in this diagram; however, in a behavior model for the User only one of these functions would be selected. All User functions instead simply serve as source or sink for information interactions with the System.

The consequence of focusing only on the behavior of one component at a time in the context of its environment is that only a limited set of use cases can be generated. For example, consider the possibility that the User in Fig. 1. does not respond to the request for unique ID. How shall the System behave then? A singular EFFBD model may be attempted to show all behaviors of all components and all corresponding interactions on the same diagram, but such a diagram would become unwieldy to read and error prone to build. The proposed alternative method employs a divide-and-conquer strategy involving the creation of a separate behavior model for each component, and specifying interactions between the components as a separate concern from that of the behavior of each component. This approach allows an architect to focus on describing behavior for one component at a time, and then separately focus on the general rules (interaction patterns appearing in all use case instances) for interaction among components. By separating these concerns, the component behaviors and interactions can be woven together during model execution, automatically generating use cases from the separate behavior and interaction specifications.
3. Monterey Phoenix: key principles and comparison with existing frameworks

Without requiring the displacement of any current notations or frameworks, MP provides complementary extensions to address questions about complex systems behaviors that are difficult to predict with existing modeling approaches. From a systems engineering point of view, the following two main principles of MP are the key for complex system and SoS behavioral analysis.

3.1. In addition to modeling the behavior of the system along with its interfaces to external systems, also model the behavior of the environment in which the system operates.

MP uses an event grammar that allows for the compact specification of behavior for each component. Events are abstractions of activities that may be experienced from the perspective of system or its environment. Data inputs and outputs are not modeled in a separate class as in EFFBD and other data flow-oriented notations, but are represented by actions (captured as events) that may be performed on that data, following the concept of Abstract Data Types (ADT) introduced in [9]. Behavior is modeled in the event grammar as an algorithm for each component, describing the step-by-step procedure by which it achieves a well-defined goal.

Events have two main binary relations used to construct event traces, or particular instances of behavior. Sequencing of events is denoted using the PRECEDES relation, and decomposition of events is denoted using the IN relation. (In data flow notations, the PRECEDES relation can often be inferred from the sequence of activities, as well as relationships of inputs/triggers and outputs among the activities.) An event grammar rule specifies the structure for a particular event type in terms of these two relations, and has form

A: right-hand-part;

where A is an event type name. More details about event grammar notation can be found in [2] and [4]. For brevity, this paper only describes the composition operations that appear in the example model.

Each component’s behavior is specified separately as a root event in the left hand part. For example, root events (lines 01 and 08) specify the behaviors of the User and the System, correspondingly. Note that the names for these root events are equivalent to the physical component names, which provides a clear link between the physical space (components) and the event
space (behaviors).

The right hand part for each of these rules comprises an algorithm for each root event. Behavior is described using composition operations such as: ordered sequence of events $A \: B \: C$, alternative $(A \: | \: B \: | \: C)$, ordered iteration $(\ast \: A \: B \: C \: \ast)$ ($A \: B \: C$ repeated zero or more times), and optional event $[A]$. For example, the User’s behavior is described on lines 02-07 below.

01 ROOT User:
02 (* request_access
03 (* creds_invalid request_access *)
04 ( creds_valid  (run_services \: | \: abandon_access_request)  |
05 creds_invalid  (attempt_exhausted \: | \: abandon_access_request )
06 end_User_session *)
07 request_access:  provide_general_ID  provide_unique_ID;

First, the user requests access (line 02). If the credentials are invalid the user repeats the request for access (line 03). Line 04 specifies what the user does when credentials are valid; the user may run services having been granted authorization, or may abandon the access request for some reason (e.g., experiences an interruption). Line 05 specifies more events that can occur when credentials are invalid: the number of allowable attempts may be exhausted (the number of access attempts is constrained in the system model), or perhaps the user may abandon the access request. The User session ends (line 06) at the conclusion of event traces for both valid and invalid credentials. In line 07, request_access is decomposed into provide_general_ID followed by provide_unique_ID, for consistency with the functions in Fig. 1. and to demonstrate the ability to create a hierarchy of events similar to a hierarchy of activities.

The System’s behavior is specified on lines 09-17:

08 ROOT System:
09 (* request_unique_ID
10  [ creds_invalid request_unique_ID
11    [ creds_invalid request_unique_ID
12      [ creds_invalid attempt_exhausted
13        invalid_creds_notice cancel_access_request ] ]
14  [ (creds_valid  ( authorize_access \: run_services |
15       long_wait_for_User cancel_access_request ) |
16       creds_invalid long_wait_for_User cancel_access_request ) ]
17 end_System_session *);

The first event in the System for this authentication scenario is request_unique_ID (line 09). If invalid credentials are supplied, the System requests the unique ID up to two more times (lines 10-11). If invalid credentials are supplied for a third time, the number of attempts are exhausted (line 12), the System provides an invalid credentials notice, and cancels the access request (line 13). If valid credentials are supplied, then the System may authorize access and run services (line 14), or cancel the access request after a long time elapses while the System is waiting for input (line 15). Yet another alternative is that invalid credentials are supplied, then there is a long wait for User input; in that case also the System will cancel the access request (line 16). Regardless of the presence or absence of valid or invalid credentials, the system will always end the session (line 17).

Note that each of these models describes events independent of interactions between the User and the System. The separation of concerns about component behavior and component interaction allows the development and synchronization of detailed algorithms for every component in the environment.

3.2. Model component interactions abstractly and separately, rather than instantiated in specific use cases.

A missing link for complex system and SoS analyses in the existing methodologies and frameworks is the concept of abstract system interactions. System interactions are often manually embedded or hard-coded into specific instances or use cases for behavior, by threading multiple components with sequenced interactions on the same diagram. For example, in Fig.1., the activity “Provide General ID” occurs before the activity “Request Unique ID.” This sequencing is accomplished in the EFFBD by the trigger “general ID.” Many use cases are slight variations of another (such as an authentication scenario resulting in success or failure), so changes to the decomposition or sequence of activities in one use case thread may trigger changes in all affected threads. One may wish to specify that in any authorization scenario, the general ID from the User always precedes a request for a unique ID from the System. In MP, this is accomplished using the COORDINATE composition operation.
This composition operation adds the PRECEDES relation between selected provide_general_ID and request_unique_ID events. The first part of the composition operation uses event patterns to specify segments of root traces that should be selected. The (* $x$: provide_general_ID *) pattern in line 18 identifies the sequence of totally ordered provide_general_ID events (with respect to the transitive closure of the PRECEDES relation). Use of the (* P *) pattern for selection means that all events P should be ordered, both iterations should have the same number of selected elements (provide_general_ID events from the first trace and request_unique_ID events (line 19) from the second), and the pair selection follows this ordering (synchronous coordination). Labels $x$ and $y$ provide access to the events selected within each iteration. The ADD composition in line 20 completes the behavior adjustment, specifying that an ordering relation will be imposed on each pair of selected events.

Likewise, one can state that the request for a unique ID from the System always precedes the providing of the unique ID from the User:

The composition operation may be considered as an abstract interaction description for root behaviors.

Note that both the User and System behavior algorithms have event names in common. A constraint must be written to explicitly state that the User and the System share all instances of those events when they occur. For example, there should be no event traces in which the access attempts have been exhausted from the User perspective but not from the System perspective—such a trace would be invalid. The SHARE ALL composition ensures that the schema admits only event traces where corresponding event sharing is implemented:

Event sharing is in fact yet another way of behavior coordination. Shared events may appear in the root event at any level of nesting.

MP is an executable architecture modeling framework. Event traces (use cases, or examples of behavior) can be generated by automated tools from the MP models. Events may be visualized as boxes, and dependencies between pairs of events as arrows marked by the relation type (Fig. 2.). Each PRECEDES relation may correspond to a control flow or trigger commonly used in flow-oriented notations (e.g., Fig.1.). Architecture views can also be extracted from MP schemas for different stakeholders to answer typical questions. The root behavior may be visualized with UML activity diagrams (see Example 7 in [4]). An MP developer’s environment may have a library of predefined views providing different visualizations for schemas.

![Example event trace (use case)](image)

Fig. 2. Example event trace (use case) where the User gets access to the System after one unsuccessful attempt. Solid arrows denote IN relations, dashed arrows depict PRECEDES relations.

4. Implementation: Using MP to purge undesired behavior from an architecture model

Using MP to automatically generate use cases from component behavior models and abstract interaction specifications, a much larger set of system behaviors can be predicted. Inspection can be used to expose design errors early in the lifecycle by
examine each generated use case for logic flaws or undesirable sequences of events that the model admits from an under-constrained specification. The true magnitude of this approach, however, is realized when combined with assertion checking, an established technique that tests for the presence or absence of specific behaviors. Jackson’s small scope hypothesis, which states that most errors can be exposed on relatively simple examples [8], provides that the scope of use case generation may be limited by simulating only a specified number of loop iterations for every event trace. MP leverages the small scope hypothesis to provide a solution to expose far more design errors than do current approaches alone, without requiring specialized skills. If an assertion results in a counterexample (an event trace that contradicts the assertion), it can be used to observe precisely why the assertion is false, and if needed, to help the architect write a constraint to prevent the sequence of events that falsifies the assertion. MP consequently provides a means for observing and correcting “unruly” behavior in a modeled architecture, so that an architect can instruct the design on what not to do, at least in specification, through the addition of abstract interaction constraints. Examples using this technique and an online demo of MP automated tools are found in [14] and [15], respectively.

5. Summary and Way Ahead

MP provides a uniform way to extract use cases from a single architecture model composed of component behavior algorithms and an abstract interaction specification—the latter being the “missing link” in current systems engineering approaches. Use cases are based on generic descriptions of system behavior, rather than on a limited number of selected use cases. This approach allows architects to expand their definitions of a “representative” set of use cases to increase the design space explored early in the lifecycle, and to correct undesired behaviors prior to the implementation. It also transfers the burden of maintaining consistency among similar use cases to automated tools.

Because of its high abstraction level, application of the MP approach should not be considered limited to the improvement of human-designed software intensive and complex adaptive systems. Design flaws manifesting at inopportune times in these classes of systems were merely the original motivation for developing this approach to behavior modeling. Future research may explore its application to the improvement of human understanding of emergent behavior in economic, biological and ecological systems, and to study the causality of events from patterns in cellular behavior to sustainable food and energy production.

Existing software engineering tools have codified the concepts described herein; the next step is to integrate them into notations and modeling environments used by systems engineers and other professionals concerned with complex technological and/or natural systems. The MP approach is a force multiplier for system architects that is open for implementation in any academic, government, or commercial modeling tool or environment whose objective involves architecting complex systems. Additional examples will be developed by the authors and interested collaborators to further test the extent of MP's capabilities in predicting complex adaptive system and SoS behavior in multiple domains.

References