Constructing A Refined Model Transition Systems In Database Using Triggered Scenarios

P VENKATA NARAYANA
M.Tech(CSE)
Department of Computer Science and Engineering
Chebrolu Engineering College
Guntur, Andhra Pradesh, India.

KONDURU NAGA NARASIMHARAO
Associate Professor
Department of Computer Science and Engineering
Chebrolu Engineering College
Guntur, Andhra Pradesh, India.

Abstract: Triggered scenarios are sequence charts that represent interactions between the system agents. Graphically a triggered scenario comprises several vertical lines labeled by names representing agent’s lifeline. Time is assumed to flow downward. Annotated arrows between these lines correspond to synchronous messages which represent instantaneous events on which both objects synchronize. In this paper, we propose a scenario-based language that supports both existential and universal interpretations for conditional scenarios. Existing model synthesis techniques use traditional two-valued behavior models, such as Labeled Transition Systems.

These are not sufficiently expressive to accommodate specification languages with both existential and universal scenarios. We therefore shift the target of synthesis to Modal Transition Systems (MTS), an extension of labeled Transition Systems that can distinguish between required, unknown, and proscribed behavior to capture the semantics of existential and universal scenarios. Modal Transition Systems support elaboration of behavior models through refinement, which complements an incremental elicitation process suitable for specifying behavior with scenario-based notations. The synthesis algorithm that we define constructs a Modal Transition System that uses refinement to characterize all the Labeled Transition Systems models that satisfy a mixed, conditional existential and universal scenario-based specification. We show how this combination of scenario language, synthesis, and Modal Transition Systems supports behavior model elaboration.

Index Terms—Scenarios; MTS; Synthesis; Partial Behavior Mode;

1. INTRODUCTION

The software engineering community has long understood the importance of requirements elicitation. Stakeholder involvement in the elicitation process and tools to help build a common ground between stakeholders and developers are essential to obtain a good requirements definition. Consequently, it is not surprising that scenarios have become increasingly popular as part of a requirements specification. Scenarios describe how system components (in the broadest sense) and users interact in order to provide system level functionality. Each scenario is a partial story which, when combined with other scenarios provides a more complete system description. Thus, stakeholders may develop descriptions independently; contributing their own view of the system to those of other stakeholders. A widespread notation for scenarios is that of message sequence charts (MSCs) and UML sequence diagrams. These notations in their most basic form are highly intuitive and have a well-understood and widely accepted semantics. However, one scenario conveys relatively little information. Many scenarios are generally required to provide a significant system description. This makes scenario synthesis—the combination of a number of scenarios into a coherent whole—a central issue. How should a set of scenarios be interpreted? How do they relate to each other?

There are two ways of tackling this issue. One is to try to infer the relations between scenarios; the other is to require these relations to be explicitly stated by stakeholders. In the latter case, what abstractions should be provided to specify these relationships? Unsurprisingly, there are many different answers to this last question. For instance, the International Telecommunication Union (ITU) introduces a graph-like notation that shows how the system evolves from one scenario to another. The underlying notion used by the ITU standard is that of scenario composition: New scenarios can be defined in terms of other scenarios by composing with sequential, choice, and iteration operators. In this way, complex system behavior can be described.

Scenario relations do not necessarily have to be given explicitly using bMSCs or state labels. Synthesis algorithms can be used to infer how scenarios are to be merged. These algorithms can include complex domain-specific and general assumptions of how scenarios are used and can sometimes incorporate additional information provided in other specifications. For example, Whittle and Schumann use the Object Constraint Language (OCL) to express pre and post conditions for messages. These are traversed with scenarios to infer from the valuation of OCL predicates how scenarios are to be related. Using this information, a state chart model is constructed for each
component of the scenario description. If the assumptions implemented in the synthesis algorithm are appropriate, they help to simplify scenario notations and reduce stakeholders’ workload. However, some drawbacks are that important explicit knowledge may be lost within the synthesis algorithms and that the consequences of the embedded assumptions can be obscured and produce misleading synthesis results. In addition, there is a significant loss of flexibility; if assumptions change, the complete synthesis procedure must be changed too. In this paper, we show how the consequences of many assumptions on how to integrate scenarios can be described explicitly in the MSC language we propose. Thus, providing a common, intermediate representation for other approaches to scenario synthesis. Regardless of the way in which the relation between scenarios is defined, the purpose of a scenario specification is to describe how a system is intended to behave. Thus, analyzing the described system behavior should play a central role in the development of scenario-based specifications. To enable such analysis, synthesis algorithms build state-machine based behavior models. In addition to providing an alternative view, there is benefit to be gained by experimenting with and replaying analysis results from behavior models in order to help correct, elaborate, and refine scenario-based specifications.

II. LITERATURE SURVEY

Composition and abstraction techniques are crucial for scalable development of modern software systems. Existing partial-behavior modeling formalisms, such as Modal Transition Systems (MTS), have the potential to support gradual refinement of a system’s early behavioral specification. In order to achieve this potential, refinement of partial behavior models should be correct in the context of hierarchical architectural specifications where (sub) system models are composed of smaller subsystems and components. Similarly, particular developmental activities will require that some details contained in the model are abstracted. In this project, with the generous help of my advisor Nenad Medvidovic.

I propose a foundational framework for reasoning about the implications of refining an MTS model that is a composition (or an abstraction) of other models. This is, to a large extent, a problem yet to be solved. The proposed framework assures that (1) a refinement of a composition can be realized with refinements of the individual composed models; and (2) a refinement of an abstract model is interpreted as a correct refinement of the detailed model. This framework is envisioned to enable techniques that support correct and scalable gradual refinement of a software system’s partial behavioral (oftentimes requirements) specification.

Software component behavior is often captured with state based models such as Labeled Transition Systems (LTS). The existing techniques for synthesis of component behavioral models from system specifications do not account for the partial nature of these specifications. I have created a synthesis algorithm for synthesizing component-level Modal Transition Systems (MTS) from use case scenarios and system-level properties. The use case scenarios are represented as a set of sequence diagrams, while the system-level properties are given in terms of OCL constraints.

The third mechanism for combining scenarios is through the use of triggers or preconditions. Instead of relating scenarios to each other, information on when each scenario can occur is provided. This approach is popular in informal development methods, where scenarios are provided with a precondition normally stated in natural language.

III. SYSTEM ARCHITECTURE

The precondition can refer informally to a state in which the scenario may occur or to a sequence of events that trigger the scenario. Other possibilities for describing preconditions are OCL or temporal logic, while scenario triggers can be specified using temporal logic or bMSC-like notations.

An advantage of using triggers is that scenarios are loosely coupled. In contrast with scenario composition methods, such as hMSCs, where the whole set of scenarios must fit together in one graph, triggers permit expression of the context of scenarios independently of existing scenarios. However, this same characteristic is a disadvantage in handling a scenario specification as a whole.

This is especially acute when triggers are the only mechanism for specifying relations between scenarios. In the work presented here, we define a scenario-based language that supports two main approaches to managing multiple scenarios, namely, those approaches based on scenario composition and on state identification. We do not support scenario triggers; although we do plan to look into this in future work.
Message Sequence Charts

Many approaches assume that a scenario describes all the concurrent behavior of participating components at a given time. This means that, for example, when the system is going through the scenario of Fig. 1, component B does not interact with other components between messages x and y. Other approaches allow further interactions on message types that do not appear in the scenario. So, for example, component B, after receiving x in Fig. 1, may be allowed to receive a message a, but not another message x before it sends message y. Other approaches allow scenarios to be composed in parallel, meaning that scenarios with common participating components can occur simultaneously.

For instance, the horizontal composition operator defined in the MSC standard has been introduced for this purpose. Composition of scenarios overlapping in time introduces a series of complex issues such as events with the same label appearing in different scenarios: Do they represent the same event and, thus, the same moment in time, or are they different occurrences of the sending or receiving of a message of the same type? In this paper, we do not consider composition of overlapping scenarios. However, this could form a future extension to our work.

IV. MTS SYNTHESIS

In this section, we define synthesis algorithms that construct behavior models in the form of Modal Transition Systems from non vacuous TSs. In general, the scenario synthesis problem consists of constructing a behavior model that satisfies a given scenario description. The problem has a number of variants depending on the scenario language used, the behavior modeling formalism chosen as a target of the synthesis, and the various additional constraints that can be imposed such as in distributed synthesis (e.g., [2]). A stronger requirement for the synthesis is that the resulting model characterizes, through some notion of refinement, all the behavior models that satisfy a given scenario description.

Synthesis Algorithm Running Example

A number of techniques that perform such synthesis have been developed. It is convenient to characterize all behavior models that satisfy a given scenario-based description in one operational model as the synthesized model can then be evolved independently of the scenario description. It can be elaborated through step-wise refinement with the guarantee that the resulting, more refined, models will continue to satisfy the scenarios. Iterative refinement can be prompted by traditional analysis techniques such as inspection, animation, and model checking.

Synthesis from eTS

We first run through an example to illustrate how an MTS characterizes all implementations that satisfy an eTS and then we present the synthesis algorithm.

The algorithm that we introduce in the next section produces the MTS in Fig. 12 (unreachable states are not shown) for the eTS discussed in the previous paragraph. All implementations of the MTS satisfy the eTS and all LTS that satisfy the eTS are implementations of the MTS. Note that in Fig. 12 states are annotated with the data structure (a tuple) that the algorithm uses to represent states. An explanation of the state’s structure will be given in Section 4.1.2. States that are not reachable from the initial state are not shown. The MTS in Fig. 12 guarantees that any of its traces that end with the sequence of actions yz lead to state 2. In other words, when the trigger of the eTS is satisfied, the MTS will be in state 2. Furthermore, note that any trace that never satisfies the trigger will only cover maybe transitions leading to states 0 and 1. That is, the MTS does not require implementations to
provide any specific behavior if the trigger of the eTS is not satisfied. From state 2, reached if and only if the trigger holds, there are two paths of required. Each path represents a word in LM. Intuitively, the state where the trigger holds has some obligations: the words in the main chart’s language. In order to make all refinements of the synthesized model satisfy the eTS we need a required path for each obligation; thus, the required transitions from (2,a,3), (3,c,4),(4,b,0), (2,a,5),(5,b,6), (6,c,0). Although states 2 through 6 have outgoing required transitions to guarantee that all implementations of the MTS will provide the behavior of the eTS’s main chart when the eTS’s trigger has occurred, these states also have maybe transitions. Where the MTS for the universal scenario differs is in the maybe transitions from states with obligations. For uTS these transitions should only allow behavior described in the main chart. The MTS in Fig. 17 only has two maybe transitions: from state 3 to 5 and back. These transitions are needed to allow LTS implementations that provide the behavior of the main chart in a deterministic fashion. Consider the LTS in Fig. 14 but in which states 3 and 5 have been joined (i.e., state 2 goes to 3=5 via a and then there is a choice on c and b to go to states 4 and 6, respectively). Such an LTS satisfies the uTS but would not be an implementation of the MTS in Fig. 17 without its transitions as the latter requires committing early to whether abc or acb will be provided while the former delays the choice until after a has occurred. Note that in the model synthesized from an eTS those maybe transitions from states with obligations also exist but they do not necessarily go to states with obligations (unless the trigger holds) as the implementations satisfying the scenario are not required to show the main chart’s behavior in every run.

V. CONCLUSION

In this paper, we have defined a scenario-specification language which includes support for describing triggered existential and universal scenarios. We have also defined a synthesis algorithm that constructs MTS models which characterize via refinement all LTS models that conform both to the existential and universal aspects of the scenario based description. A novel aspect of the approach is the use of triggered existential scenarios which have a branching semantics. This is in line with existing informal scenario-based and use case-based approaches to requirements engineering exploiting the expressive power of MTS in an operational behavior model.

VI. REFERENCES


