From System Specifications to Component Behavioral Models

Ivo Krka, George Edwards, Yuriy Brun, and Nenad Medvidovic
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781, USA
{krka,gedwards,ybrun,neno}@usc.edu

Abstract

Early system specifications, such as use-case scenarios and properties, rarely completely specify the system. Partial models of system-level behavior, derived from these specifications, have proven useful in early system analysis. We believe that the scope of possible analyses can be enhanced by utilizing component-level partial models. In this paper, we outline an algorithm for deriving a component-level Modal Transition System (MTS) from system-level scenario and property specifications. The generated MTSs capture the possible component implementations that (1) necessarily provide the behavior required by the scenarios, (2) restrict behavior forbidden by the properties, and (3) leave the behavior that is neither explicitly required nor forbidden as undefined. We discuss how these generated models can help discover system-design flaws, support requirements elicitation, and help select off-the-shelf components.

1 Introduction

Design activities that take place during the early phases of system development, such as elicitation, definition, and refinement of requirements, occur in the context of incomplete information and uncertainty. These activities benefit from, and also result in, *partial* system specifications that capture high-level system behavior while deferring many decisions to subsequent design phases. Two commonly used types of such specifications are (1) *scenarios* (e.g., UML sequence diagrams) that model examples of important system use cases, and (2) *properties* (e.g., OCL constraints) that model conditions and constraints on as well as relationships among system elements.

Scenarios and properties form a basis for gradual definition and refinement of system requirements and architecture, and can be used for early system analysis (e.g., discovery of design defects and conflicting requirements). The utility of scenarios and properties arises from their ability to express partial but straightforward views of the system that are useful for communicating intent among stakeholders.

Numerous existing techniques [4, 9, 10, 11] leverage scenario and/or property specifications to derive concrete behavioral models of system components that can be leveraged for architectural analysis and assessment. However, these approaches ignore the partial and limited nature of such specifications by attempting to generate complete behavioral models. Instead, we argue that designers need formal models that describe systems in terms of (1) behavior *required* by the scenarios, (2) behavior *prohibited* by the constraints, and (3) *potential* behavior that is neither required nor forbidden by the specifications.

Recently, researchers have developed automated processes for deriving Modal Transition Systems (MTSs) [3], a partial behavior modeling formalism, from system scenarios and properties [6, 7]. An MTS describes the behavior of a system via states with transitions, which are labeled as either *required* or *potential* transitions. MTSs have proven useful in requirements elicitation, architectural refinement [6, 7], and verification that a system implementation satisfies specified properties [2].

Current approaches to MTS derivation adopt a system-wide view and synthesize only system-level MTSs. However, modern systems are typically built from independent components, and behavioral models that exploit a component-oriented perspective are required for rigorous architectural analysis. We are developing algorithms and methods for (1) deriving component-level MTSs from system-level scenarios and properties, and (2) leveraging component-level MTSs in system development.

In this paper, we outline our algorithm for component-level MTS generation from system scenarios and properties (Section 3) and describe how our algorithm can help discover system-design flaws (Section 4). We also envision component-level MTSs (1) assisting in requirements elicitation, (2) supporting selection of candidate off-the-shelf components, and (3) enabling comparison of as-intended and as-implemented systems.

ICSE'09, May 16–24, 2009, Vancouver, Canada
978-1-4244-3494-7/09/$25.00

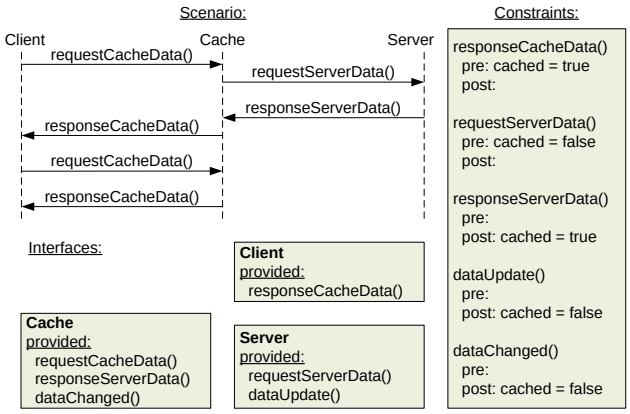

Figure 1. Scenario, property, and component interface specifications for a web cache system.

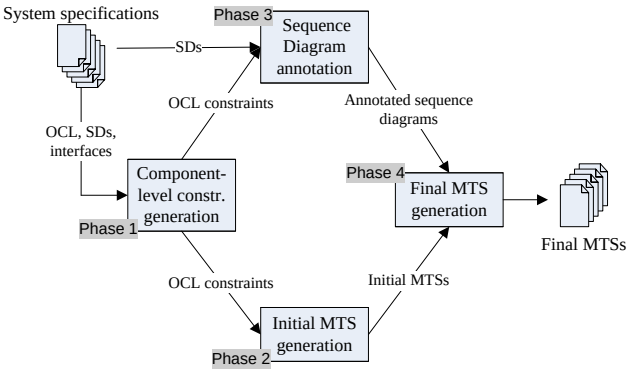

Figure 2. Phases of the MTS-generation algorithm

2 Background and Related Work

In order to ease the adoption of our techniques, we employ previously defined specification formalisms that are already in widespread use [3, 5]. We use basic UML sequence diagrams [5] to depict scenarios and OCL [5] to capture intended system properties. Like other research on behavioral models [8, 10], we leverage only a subset of OCL: conjunctive (AND) clauses of Boolean expressions on system variables that define pre- and postconditions of system operations. Figure 1 illustrates an example specification of a web cache system (described in further detail in Section 3).

The MTS [3] formalism extends on the LTS formalism [9] to permit modeling of potential transitions (in addition to required transitions). The *strong refinement* relation [1] of MTSs results in the conversion of potential transitions into required transitions or removal of the potential transitions, based on new knowledge about system behavior. MTS N is a refinement of MTS M if and only if every required transition in M is also required in N and every potential transition in N is also potential in M. We refer the reader to [1] for formal, complete definitions of MTSs and the strong refinement relation.

In related work, Whittle and Schumann [10] proposed an algorithm to generate component statecharts from scenarios and properties. Mäkinen and Systä [4] developed a semi-automated approach for synthesizing statecharts from sequence diagrams with architect guidance. The method of Ziadi et al. [11] synthesizes component statecharts from sequence diagrams with complex constructs (such as references, loops, and forks). Finally, Uchitel et al. [9] put forward a technique to generate component-level LTS models and compose those models to discover implied scenarios. All four of the above approaches presume complete system specification, which is not realistic during early design [8].

The work we present here addresses this problem by generating partial component-level behavioral models from partial system-level specifications. Furthermore, the main conceptual distinction between our work and the work done in [6, 7] is that we generate component-level MTS models as opposed to the system-level MTS models. By moving the focus from the system-wide perspective to a component-level perspective, we enable the benefits outlined in Section 1.

3 Generating Component-Level MTSs

In this section, we outline our algorithm for generating component-level MTSs from scenario and property specifications. Figure 2 depicts four phases of the algorithm. The inputs to the algorithm are UML sequence diagrams describing system execution scenarios and OCL constraints defining pre- and postconditions on component operations. The scenarios can be defined on different sets of component operations, while the OCL constraints are defined on a set of system state variables. Additionally, we assume the availability of provided interface specifications for system components, e.g., such as those typically provided in UML class diagrams.

The algorithm first derives component-level constraints (Phase 1), which are later used to define the state space of generated MTSs (Phase 2) and derive the conditions on system state variables under which scenarios can execute (Phase 3). The returned set of component-level MTSs (Phase 4) requires all behavior specified in scenarios and allows behavior that does not conflict with component-level constraints as potential behavior.

Figure 1 depicts an example specification for a simple web caching system. In the given scenario, a *Client* requests data from the *Cache*, which then fetches the data from the *Server*. The *Cache* handles the subsequent request from the

Client. We now provide the functional descriptions of the phases of the algorithm.

Phase 1: Component-level constraint generation. The first phase of our algorithm translates the system-level constraints, defined on system state variables, into constraints on the behavior of each component. First, using the OCL constraints, we determine which system state variables (called component's relevant state variables) constrain invocations of each component's required operations. Second, for each component, we determine which constraints (called the component-level constraints) contain only the expressions from the pre- and postconditions of that component's interface operations defined on that component's relevant state variables. For example, the *Client* from Figure 1 does not have any relevant state variables as the specifications do not constrain the *Client* in sending requests to *Cache*. *Cache*, however, has $cached$ as the relevant state variable because *Cache*'s behavior is constrained by the value of $cached$. *Cache*'s component-level constraints are thus defined on $cached$.

Phase 2: Initial MTS generation. The second phase of our algorithm creates *initial component MTSs* that capture all potential behaviors of components, i.e., behaviors that are not proscribed by component-level constraints from Phase 1. We construct the initial MTS for a component *A* as follows. First, we create a set of states, such that each state is assigned a unique truth assignment of the expressions in *A*'s component-level constraints. We then add a potential transition labeled o from each state S to each state P if and only if the preconditions of o hold in S and the postconditions of o hold in P. For example, the initial MTS of the *Cache* has a potential transition labeled $responseServerData$ from state $cached$ = false to state $cached$ = true.

Phase 3: Sequence diagram annotation. The third phase of our algorithm derives the conditions under which the given scenarios can execute. The scenario specifications convey information about the sequences of events the system should exhibit, but usually do not provide information about the conditions under which these sequences can actually execute. We address this issue in a manner similar to Whittle et al. [10]: for each component, we annotate that component's interactions in each scenario with the truth assignments of component-level constraint expressions that must be satisfied before and after the interaction occurs. For instance, this phase of our algorithm discovers that, from *Cache*'s perspective, $cached$ must initially be false but must ultimately be true in the given scenario. This phase of the algorithm also discovers discrepancies between the scenario and property specifications.

Phase 4: Final MTS generation. The last phase of our algorithm produces desired component-level MTSs by combining the initial MTSs and the annotated sequence diagrams. For component *A*, we refine potential behavior in the initial MTS by adding required behavior captured in the corresponding annotated scenario. A potential transition from state S to state P on operation o is changed to a required transition if the scenario annotation (i.e., expression truth assignments) before o is satisfied in S, and the scenario annotation after o is satisfied in P. We also refine state P into two new states P_1 and P_2, each one accounting for a particular subset of P's incoming transitions. Both P_1 and P_2 have the same outgoing transitions as P. Figure 3 shows the final MTS for the *Cache* component with potential transitions marked with "?" and required transitions in bold. There are no required transitions for operation $dataChanged$ as this operation does not appear in the scenario specification. Further, from the final MTS, we can observe that $cached$ changes from false to true during the $responseServerData$ operation.

4 Utilizing Component-Level MTSs

In this section, we outline the design flaws our MTS generation process can discover and discuss potential uses of component-level MTSs. Component-level MTSs can facilitate a number of benefits, including: (1) requirements elicitation, (2) selection of candidate OTS-components, and (3) comparison of designed and implemented components.

Discovery of specification discrepancies. The annotation of sequence diagrams (Phase 2) directly exposes two types of potential problems in system specifications: conflicts between the specified scenarios and properties and the presence of inconsistent component states. Discrepancies between scenarios and properties occur when constraints prohibit the sequence of operations described in a sequence diagram. A discrepancy may be the result of either overly restrictive constraints or misspecified scenarios. Inconsistent states arise when multiple components' expected values for a system state variable differ.

Requirements elicitation. Eliciting requirements from system-level MTSs, as in [7], can introduce design flaws because requirements gathered using a system-level perspective can conflict with what is implementable at the component level. For example, a new scenario extracted from a system-level MTS might only be possible when all participants have global knowledge. However, this assumption does not hold in component-based systems without incurring significant synchronization overhead. On the other hand, eliciting requirements using component-level MTSs provides some assurance that inconsistent assumptions are not introduced. In the web cache system, the *Cache* MTS can serve as the basis of new requirements such as: (1) the $dataChanged$ operation may be invoked at any time, and (2) $requestServerData$ shall be invoked only when there is a pending *Client* request.

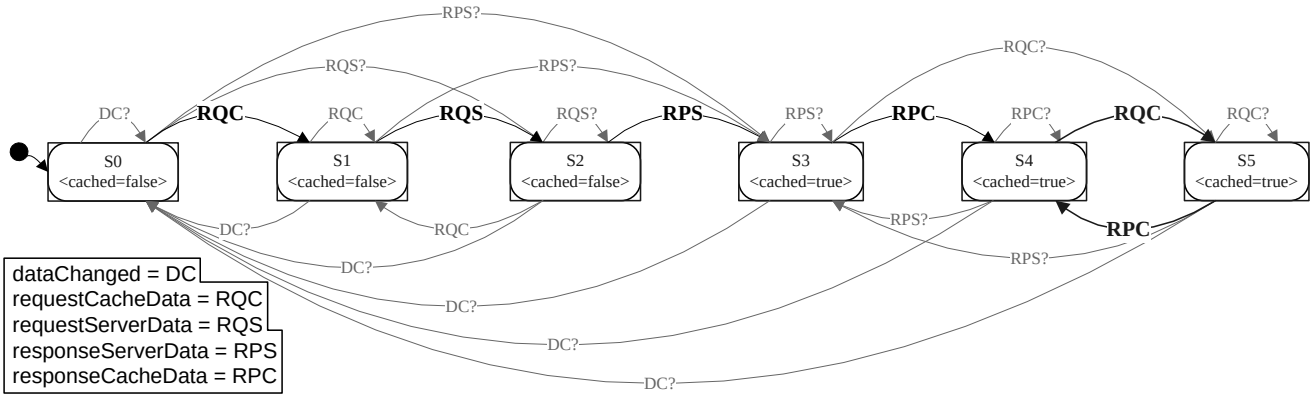

Figure 3. Generated component-level MTS for a web cache component

Off-the-shelf (OTS) component selection. Modern component-based systems rely heavily on reuse. However, OTS-component selection processes are often based on ad-hoc and labor-intensive analyses of stakeholder interests. Component-level partial behavioral models can guide the OTS-component selection process because they capture the required behavior that must be implemented in the final system and the potential behavior that is neither explicitly required nor prohibited by the requirements. Component-level MTSs may thus be used to mine OTS-component repositories for suitable components that exhibit the required behavior and possibly provide a subset of the potential behavior.

As-intended vs. as-implemented system comparison. Generated component-level MTSs capture architects' intent, as defined in scenarios and properties. However, the implemented system might differ significantly from those intentions. While comparing whole-system behavior to the specification is a complex task, component-level comparison may be far simpler. Further, identification of differences between the specification and the implementation may assist analysis of potential risks associated with providing functionality that was not originally intended and/or removing or modifying functionality that was nominally required.

5 Summary

Generating component-level MTSs from system-level scenario and property specifications detects discrepancies within the specifications, assists further requirements elicitation, facilitates OTS-component selection, and can verify the consistency of a component's implementation with its specification. Our planned next steps are to (1) prove our algorithm's correctness by ensuring that it does not create conflicts with the specifications under any circumstances, (2) evaluate its utility on real-world systems, also comparing it with other model synthesis techniques, and (3) implement tool support for creating and utilizing component-level MTSs. We will assess the tractability and usability of all four ways of utilizing MTSs described in this paper. We envision the automated generation of component-level MTSs greatly influencing the early system development processes.

References

[1] D. Fischbein and S. Uchitel. On consistency and merge of modal transition systems. In *Proc. of FSE*, 2008.

[2] P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based model checking using modal transition systems. In *Proc. of CONCUR*, 2001.

[3] K. G. Larsen and B. Thomsen. A modal process logic. *Logic in Computer Science*, 1988.

[4] E. Mäkinen and T. Systä. MAS — an interactive synthesizer to support behavioral modelling in UML. In *Proc. of ICSE*, 2001.

[5] D. Pilone and N. Pitman. *UML 2.0 in a Nutshell*. O'Reilly Media, Inc., 2005.

[6] G. Sibay, S. Uchitel, and V. Braberman. Existential live sequence charts revisited. In *Proc. of ICSE*, 2008.

[7] S. Uchitel, G. Brunet, and M. Chechik. Behaviour model synthesis from properties and scenarios. In *Proc. of ICSE*, 2007.

[8] S. Uchitel, J. Kramer, and J. Magee. Behaviour model elaboration using partial labelled transition systems. In *Proc. of FSE*, 2003.

[9] S. Uchitel, J. Kramer, and J. Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM TOSEM*, 13(1), 2004.

[10] J. Whittle and J. Schumann. Generating statechart designs from scenarios. In *Proc. of ICSE*, 2000.

[11] T. Ziadi, L. Helouet, and J.-M. Jezequel. Revisiting statechart synthesis with an algebraic approach. In *Proc. of ICSE*, 2004.