Synthesizing Partial Component-Level Behavior Models from System Specifications

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

ABSTRACT

Initial system specifications, such as use-case scenarios and properties, only partially specify the future system. We posit that synthesizing partial component-level behavior models from these early specifications can improve software development practices. In this paper, we provide a novel algorithm for deriving a Modal Transition System (MTS) for individual system components 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 also show how our algorithm helps to discover potential design flaws.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Requirements/Specifications; D.2 [**Software Engineering**]: Software Architectures

General Terms

Algorithms

Keywords

behavior model synthesis, Modal Transition Systems, scenarios, constraints, partial specifications

1. INTRODUCTION

Early software development activities, such as elicitation and refinement of requirements, and preliminary design, occur in the context of incomplete information and uncertainty. These activities benefit from, and also result in, *partial* specifications that capture high-level system behavior while deferring many decisions to subsequent development phases. Two commonly used types of such specifications are (1) *scenarios* that exemplify important system use cases and module interactions (e.g., UML sequence diagrams), and (2) *properties* that model conditions and constraints on system elements (e.g., OCL constraints). Scenario and property models are focused and straightforward but partial views of a system: they do not describe all functionality that the system will provide nor how that functionality will be implemented. This feature of scenarios and properties allows architects to defer decisions about some aspects of the system, cope with ambiguous requirements, and gradually refine the system requirements and architecture.

Software architects often utilize the available set of scenarios and properties in order to refine the architectural design and produce component-oriented behavior models. Component-level models are required for rigorous architectural analysis and form a basis for subsequent implementation. Numerous techniques (e.g., [4, 6, 19, 21]) leverage scenario and/or property specifications to derive behavior models of system components. However, these approaches produce final models of component behavior and ignore the partial nature of the input specifications [18]. We argue that designers need formal models that precisely, while partially, describe system components in terms of (1) the behavior *required* by the scenarios, (2) the behavior *prohibited* by the constraints, and (3) the *potential* behavior that is neither required nor forbidden by the specifications. The architect must then decide while performing the architectural refinement whether the potential behavior should become required or prohibited.

Other recently proposed approaches for behavior model synthesis from scenarios and properties [17, 20] account for the inherent partiality of these specifications by representing system behavior with Modal Transition Systems (MTS). MTSs are both formal, in the sense that they have precisely-defined semantics, and partial, in that they permit the inclusion of behaviors about which only limited information is yet available. MTSs represent behavior as a set of states and a set of transitions between those states. Each transition may be either a *required* or a *potential* transition.

Current MTS-synthesis approaches have several shortcomings:

1. They produce system-level, as opposed to component-level, models, implicitly assuming that each component has global knowledge of the system's execution.
2. They generate models only from a single narrowly focused specification formalism (e.g., epLSC [17]).
3. They make restrictive assumptions about the specifications (e.g., each scenario is executable from the initial state [20]).
4. They scale poorly with system size, resulting in prohibitively large system-level models.

In this paper, we give a detailed description of our algorithm for generating *component-level* MTSs from system specifications. In our earlier short paper, we outlined the intuition behind the algorithm [11]. Our approach is directly motivated by the above observation that architects need formal, component-oriented models that capture the inherent partiality of early specifications. We derive component-level MTSs from (1) system-level scenarios, captured as UML sequence diagrams, and (2) system-level properties, captured as OCL constraints. Our algorithm directly addresses

ESEC-FSE'09, August 24–28, 2009, Amsterdam, The Netherlands.

the aforementioned shortcomings of the existing behavior model synthesis approaches. We first translate the provided system-level properties to behavioral restrictions on individual components. For each component, we generate an MTS that captures component behaviors that are not prohibited by the derived component-level properties. We then extract the conditions under which provided scenarios execute and refine the initial MTS with the behavior from the scenarios. We also show how artifacts generated using our algorithm expose potential design flaws, including misspecified scenarios, overly restrictive properties, and inconsistencies between the system and component viewpoints.

The rest of the paper is organized as follows. In Section 2 we provide the background on the utilized specification and behavior modeling formalisms; we also introduce a running example. In Section 3, we provide a detailed description of the individual phases of our component-level MTS synthesis algorithm. We evaluate the algorithm by (1) analyzing the complexity of the algorithm's phases (Section 3) and (2) showing how the generated artifacts facilitate the discovery of specification discrepancies and potential design flaws (Section 4). We overview the related work in Section 5 and conclude with Section 6.

2. BACKGROUND

In this section, we describe the notations we use to specify scenarios and properties (Section 2.1) and formally define the formalisms we use to capture the behavior of software components (Section 2.2). We also introduce an example system leveraged in this paper (Section 2.3).

2.1 Scenarios and Properties

System scenarios and properties, generated during requirements elicitation or obtained from the domain knowledge, may be specified in a variety of ways. For this task, we leverage two commonly used formalisms: (1) UML Sequence Diagrams (SDs) [16] to specify the use case scenarios, and (2) OCL constraints [16] to capture the system properties. Figure 1 presents the Web cache system specified using these formalisms.

In general, SDs consist of lifelines that represent runtime component instances participating in a scenario[1], and arrows between the lifelines that represent component interactions. For clarity and brevity, the discussion in this paper is limited to SDs without advanced constructs; however, our algorithm is capable of handling complex constructs such as (1) SD reference (by replacing the reference with the actual sequence), (2) parallelism and conditionals (by creating a separate SD for each possible execution sequence), and (3) loops (we discuss loops in Section 3).

As in previous work [18, 21], we define pre- and postconditions on individual system operations as conjunctive clauses of Boolean expressions on high-level system domain variables. Although we showcase our approach on OCL constraints representable in propositional logic, where the domain variables are mapped to atomic propositions, the presented concepts are generally applicable to the subset of OCL that can express first-order logic terms. We believe that the high level, abstract nature of properties specified during early development phases makes them well-suited for specification via first-order relations on predicates expressed in OCL.

2.2 Modal Transition Systems

A Labeled Transition System (LTS) [14] is a state machine-based formalism that labels transitions to specify which actions can occur

[1]If SD lifelines do not map to distinct components, the architect should provide this mapping in order to use our algorithm.

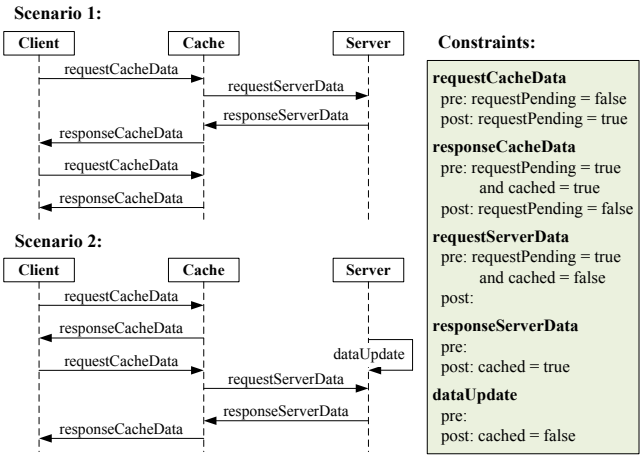

Figure 1: Web cache specification.

in each state. LTSs are often used to model the required behavior of a software component. A Modal Transition System (MTS) [12] is a generalization of LTS that allows modeling of transitions that are neither explicitly required nor prohibited in particular states as *potential* transitions. An example MTS depicted in Figure 5 (discussed in more detailed later in the paper) contains only potential transitions which are marked with '?' at the end of the labels. For instance, this means that it is not yet known whether $responseServerData$ should be required or prohibited in s_0.

DEFINITION 1 (LABELED TRANSITION SYSTEM (LTS)). *A labeled transition system L is a tuple (S, A, Δ, s_0), where S is a finite set of states, A is a finite set of labels, $\Delta \subseteq (S \times A \times S)$ is a transition relation, and s_0 is the initial state.*

DEFINITION 2 (MODAL TRANSITION SYSTEM (MTS)). *A modal transition system M is a 5-tuple $(S, A, \Delta^r, \Delta^p, s_0)$, where $\Delta^r \subseteq \Delta^p$, (S, A, Δ^r, s_0) is an LTS comprising the required transitions and (S, A, Δ^p, s_0) an LTS comprising potential transitions.*

The strong refinement relation of MTSs [7] captures the notion of converting potential transitions into required transitions or removing them from the model, based on newly available information. An MTS N is a strong refinement of an MTS M if N has all the required behavior of M, some required behavior that was previously only potential in M, and potential behavior that was also potential in M. We use the notation $s \xrightarrow{l}_r s'$ if there is a required transition from state s to s' labeled l, and $s \xrightarrow{l}_p s'$ if there is a potential transition from state s to s' labeled l.

DEFINITION 3 (STRONG REFINEMENT RELATION). *For all MTSs M and N, a strong refinement relation R is a binary relation $R \subseteq S_M \times S_N$, $(s_{M0}, s_{N0}) \in R$, such that for each pair $(s_M, s_N) \in R$:*

1. $(\forall l, s'_M)(s_M \xrightarrow{l}_r s'_M \Rightarrow (\exists s'_N)(s_N \xrightarrow{l}_r s'_N \wedge (s'_M, s'_N) \in R))$

2. $(\forall l, s'_N)(s_N \xrightarrow{l}_p s'_N \Rightarrow (\exists s'_M)(s_M \xrightarrow{l}_p s'_M \wedge (s'_M, s'_N) \in R))$

We say that MTS N refines MTS M if there exists a strong refinement relation R from M to N.

2.3 Running Example

We leverage the simplified Web cache system depicted in Figure 1 to illustrate important concepts throughout the paper. As mentioned earlier, the system specification consists of a set of UML SDs and OCL pre- and postconditions on system operations.

In Scenario 1 from Figure 1, Client requests data from Cache, Cache fetches the data from Server and returns it to Client. A

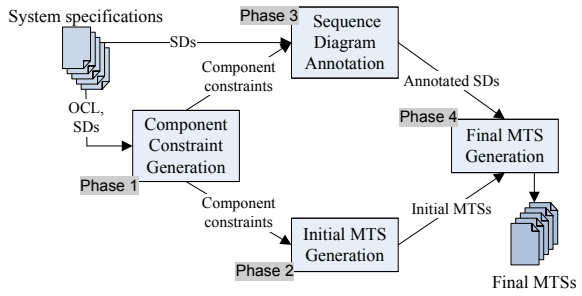

Figure 2: MTS synthesis algorithm phases.

subsequent request is then directly retrieved from Cache. In the other scenario, Client asks for data that is already $cached$, and the data is directly returned. Next, a *dataUpdate* occurs (indicating that the data has changed), and a subsequent data request is redirected to Server. Similarly, the constraints in Figure 1 imply that Cache requests data from Server only if there is a pending client request ($requestPending = true$) and the data is not cached ($cached = false$). It is important to note that these variables are system-level, rather than component-level, domain variables.

3. SYNTHESIZING MTS MODELS

This section describes our algorithm that automatically generates a component-level MTS for each component in a software system from a set of scenario and property specifications. As we will demonstrate, the generated MTSs — in addition to being useful as a basis for traditional detailed design and analysis activities — can also be used to discover discrepancies between system-wide and component-centric perspectives. These discrepancies often represent flaws in the system's design.

We assume that a system's early specifications are available in the form of (1) a set of UML SDs and (2) a set of OCL constraints specified on system domain variables. We expect that software architects and domain experts are generally able to provide such specifications. The presented approach is also well suited in case of incremental provision of specifications, since additional information will mainly only refine the previously synthesized model.

The final result of our algorithm is a set of component-level MTSs that capture the behavior required by the specifications and do not allow the behavior proscribed by the specifications. Several important obstacles must be overcome in order to generate the component-level MTSs:

1. Obtaining constraints on the behavior of individual components, when input specifications are given at the level of the whole system.
2. Constructing the state space of each component and determining all potential transitions that are not proscribed by the provided system specification.
3. Determining the conditions under which a specified scenario can execute from each component's perspective.
4. Incorporating information about required system behavior, which is captured in system scenarios, into the partial behavior models of components.

Our algorithm consists of four distinctive phases, as depicted in Figure 2. Each phase explicitly addresses one of the above challenges. The following sections describe the phases of our algorithm in detail. We evaluate the worst-case time complexity and the complexity expected in practice of the algorithm's phases. Additionally, we formally prove that Phase 4 of the algorithm ensures the strong refinement relation [7] between the initial and the final MTSs.

Phase 1: Component Constraint Generation

The first phase of our algorithm produces a set of OCL constraints on the behavior of each component in the system. These constraints are used in later phases of the algorithm to create initial component MTSs and annotate SDs. For each component C, we derive OCL constraints that define pre- and postconditions on both the provided and expected operations of C. The major steps in this phase of the algorithm are described next.

Derive provided and required operations.

In our approach, we do not assume the availability of a component's operation signatures; hence we first extract the set of component operations from the available specifications. We define the set P_C of component C's provided operations and the set E_C of C's expected operations as follows:

DEFINITION 4 (PROVIDED OPERATIONS). *For every component C, for all operations o, $o \in P_C$ iff there exists an SD with operation o labeled on an incoming arrow into C's lifeline. We call P_C the set of C's* provided operations.

DEFINITION 5 (EXPECTED OPERATIONS). *For every component C, for all operations o, $o \in E_C$ iff there exists an SD with operation o labeled on an outgoing arrow from C's lifeline. We call E_C the set of C's* expected operations.

For example, *responseCacheData* is Client's provided operation, while *requestCacheData* is its expected operation (see Figure 1).

Derive significant domain variables.

As noted earlier, our approach is intended for early development phases, when explicit information about the behavior of a component is not available. In order to derive a component's behavior model, we leverage domain variables. Specifically, we determine component states based on the value assignments of domain variables that affect that component's behavior — different states will have different value assignments. Additionally, assigning values to domain variables for each state helps us determine whether a component is allowed to invoke an operation when in a particular state.

Not every domain variable affects the behavior of a component. For example, the value of domain variable $cached$ does not affect the behavior of Client in the Web cache system (recall Figure 1): Client will send data requests whether or not the data is cached. Therefore, we must determine which domain variables restrict the behavior of a specific component. For a component C, we first determine which domain variables restrict C's outgoing behavior and refer to these variables as C's significant domain variables.

DEFINITION 6 (SIGNIFICANT DOMAIN VARIABLES). *Let V be the set of system domain variables. For each component C, $V_C \subseteq V$ is the set of* significant domain variables *iff for all $v \in V_C$, v appears in a precondition of an expected operation in E_C.*

For Cache in the Web cache system, $cached$ and $requestPending$ are Cache's significant domain variables because Cache should invoke *requestServerData* and *responseCacheData* only depending on the values of these variables.

Derive scoped domain variables.

To complete a component's behavior model, we need to determine for every component state whether any provided operations may be invoked while the component is in that state. We utilize the system specifications to infer the conditions (in terms of domain variable values) that have to hold for a provided operation to be invoked. For a component C, we first determine the set of domain variables that are only modified by operations in which C participates. We term these *scoped domain variables* because they exist globally inside C's "scope": these variables are modified only by

```
GenCompConstr(constr Cons, comp C)
1   create new empty constraint set Cons_C
2   for each operation o in E_C
3     for each expression e in pre_o
4       if e is defined on any variable in V_C
5         Cons_C.add(o.pre, e)
6     for each expression e in post_o
7       if e is defined on any variable in V_C ∪ U_C
8         Cons_C.add(o.post, e)
9   for each operation o in P_C
10    for each expression e in pre_o
11      if e is defined on any variable v in U_C
12        Cons_C.add(o.pre, e)
13    for each expression e in post_o
14      if e is defined on any variable in V_C ∪ U_C
15        Cons_C.add(o.post, e)
16  return Cons_C
```

Figure 3: Deriving component-level constraints.

C's operations. The constraints that specify when C's provided operations may be invoked (from C's perspective) are defined in terms of scoped domain variables.

DEFINITION 7 (SCOPED DOMAIN VARIABLES). *Let V be the set of system domain variables. For each component C, let U_C be the set of* scoped domain variables*, such that $v \in U_C$ iff v is modified only by the operations $o \in P_C \cup E_C$.*

For instance, $requestPending$ is Client's only scoped domain variable; Client has global knowledge of $requestPending$'s value.

Translate system-level to component-level constraints.

After obtaining the sets V_C and U_C, we can derive component-level constraints from the available system-level constraints. The system-level constraints cannot be applied as-is because, as previously stated, not every domain variable affects a component's behavior. We thus leverage the knowledge of a component C's significant and scoped domain variables to decide which subexpressions of the operation pre- and postcondition are relevant from C's perspective. For example, Client's component-level constraints (Figure 4) contain the precondition $requestPending = true$ for *responseCacheData*, which differs from *responseCacheData*'s system-level precondition (Figure 1) because $cached$ does not restrict the behavior of Client. Pseudocode in Figure 3 describes, in detail, the steps for obtaining the component-level constraints. In the rest of the paper we refer to both the component C's significant domain variables and scoped domain variables as C's *significant variables*.

Complexity analysis: Let N_C be the number of components in the system, N_{SD} be the number of SDs, N_{OP} be the number of distinct operations, N_V be the number of domain variables, L_{SD} be the maximum length of an SD, N_{SV} be the maximum number of component's significant variables, and N_{COP} be the maximum number of component's operations (we will use these symbols in our complexity analysis of each of the algorithm phases). In Phase 1, we make a single pass through each SD and two passes through each operation's constraints. Thus, the worst-case time complexity for this phase is $\Theta(N_{SD} \times L_{SD} + N_C \times N_{OP})$. In practice, we expect N_{SD} and N_{OP} to be the largest factors, ranging to up to several hundred, so this phase of the algorithm will execute in time linear in the number of SDs and distinct operations.

requestCacheData
pre: requestPending = false
post: requestPending = true

responseCacheData
pre: requestPending = true
post: requestPending = false

Figure 4: Client's component-level constraints.

Phase 2: Initial MTS Generation

After deriving the set of component-level constraints for each component C, we generate an initial MTS M_C that captures all the possible behavior that is not proscribed by these constraints. We capture this behavior as potential transitions labeled with the operation names. In this section, we describe the steps performed to create the set of initial component-level MTSs.

Extend the MTS definition.

Earlier, we pointed out that we distinguish component's behavior states based on the value assignment of their significant variables. Additionally, during construction of the initial MTS we need to decide whether or not a transition between two states should be added to the MTS. To effectively address these requirements, we extend the earlier definition of an MTS.

DEFINITION 8 (MTS DEFINITION EXTENSION). *Extended MTS M_C is a structure (S, A, Δ_r, Δ_p, s_0, T_C, Map), where (S, A, Δ_r, Δ_p, s_0) is an MTS as specified in Definition 2, T_C is the set of vectors of possible truth assignments for C's significant variables, and function $Map\colon S \rightarrow T_C$ maps each state in M_C to a vector of truth assignments for C's significant variables.*

For example, consider Cache in the Web cache system. T_{Cache} has vectors ⟨false, false⟩, ⟨false, true⟩, ⟨true, false⟩, and ⟨true, true⟩. These combinations correspond to the possible truth assignments of significant variables $requestPending$ and $cached$.

Create an initial state.

The first step in the construction of an initial MTS is defining the starting MTS state s_0 and setting $Map(s_0)$ to the initial values of the component's significant variables. The initial values can typically be extracted from the system requirements or otherwise obtained from a domain expert. For example, the initial state of Cache's MTS (Figure 5) has the corresponding significant variable value mapping $Map(s_0) = \langle$false, false$\rangle$ pertaining to initial values of $requestPending$ and $cached$.

Expand the MTS with new transitions and states.

We do not know a priori all the states that comprise the state space of a component's MTS. Therefore, after creating the starting MTS state, we gradually construct the desired MTS by (1) adding new transitions from existing states and (2) adding new states if the transition should lead to a state that is not already in the MTS. To determine which transitions are allowed from a particular state, we leverage the definition below.

DEFINITION 9 (ALLOWED MTS TRANSITIONS). *A transition $s \xrightarrow{o}_p s'$ labeled with the name of the operation op is allowed iff (1) the op's precondition is satisfied in s (i.e., for $Map(s)$) and*

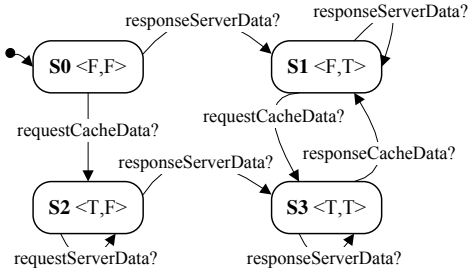

Figure 5: Cache's initial MTS. States are marked with values of ⟨requestPending,cached⟩.

```
GenInitialMTS(comp C, cons Cons_C, vec init)
1   create new MTS M_C
2   create initial state s_0 with Map(s_0) = init
3   M_C.S.add(s_0)
4   iterator it = M_C.S.iterator()
5   while iterator.hasNext() ≠ NULL
6     s_curr = iterator.nextElement()
7     for each operation o ∈ R_C ∪ P_C
8       if Map(s_curr) satisfies C.pre_o
9         {let S_next be a set of states s for which
10        transition s_curr -o->_p s is allowed}
11        for each s ∈ S_next
12          if s ∉ M_C.S
13            M_C.S.add(s)
14          M_C.Δ^p.add(s_curr -o->_p s)
15  return M_C
```

Figure 6: Generating an initial MTS.

```
Annotate(SD SD_C, comp C)
1   for each operation invocation i ∈ S_C
2     {create two new vectors an_{C,i} and an'_{C,i} indexed
3     by C's significant variables and initialized to ?}
4   for each C's significant variable v
5     if v must be assigned a value x to satisfy C.pre_i
6       an_{C,i}[v] = x
7     if v must be assigned a value x to satisfy C.post_i
8       an'_{C,i}[v] = x
9     if an_{C,i}[v] = ?  and an'_{C,i}[v] ≠ ?
10    and an_{C,i}[v] = an'_{C,i}[v] is not a requirement
11      an_{C,i}[v] = ⋆
12    if an'_{C,i}[v] = ?  and an_{C,i}[v] ≠ ?
13    and an_{C,i}[v] = an'_{C,i}[v] is not a requirement
14      an'_{C,i}[v] = ⋆
15  return SD_C
```

Figure 7: The initial SD annotation steps.

op's postcondition is satisfied in s', and (2) s and s' have identical values for those significant variables that are not modified by op.

Hence, we add the transitions that are allowed from existing states to the MTS under construction, and add their destination states s' to the MTS if these are not already contained in the state set. By following these steps, we ensure that we do not create any transitions that violate the constraints. The generated MTS can contain non-deterministic transitions since we do not explicitly model transition guards. Reasoning about non-determinism should thus be performed by the architect. The pseudocode in Figure 6 details the steps we perform to generate an initial MTS for a component.

For example, in Cache's initial MTS, starting with state s_0, we expand the MTS with two transitions: $s_0 \xrightarrow{requestCacheData}_p s_2$ and $s_0 \xrightarrow{responseServerData}_p s_1$, because the preconditions of *requestCacheData* and *responseServerData* are satisfied in s_0 (see Figure 5). Further, we add new state s_1, which preserves s_0's value of $requestPending$ and satisfies *responseServerData*'s postcondition; and state s_2, which preserves s_0's value of $cached$ and satisfies *requestCacheData*'s postcondition. Additional expansion steps result in the complete MTS from Figure 5.

Complexity analysis: In Phase 2, we gradually build the state space and add necessary transitions to the initial MTSs. The worst-case time complexity for this phase is $\Theta\left(2^{N_{SV}} \times N_{COP} \times N_C\right)$. Although exponential, this complexity should not be problematic in practice for several reasons. First, N_{SV} will be reasonably small because, in practice, a component will be concerned only with a subset of domain variables, and N_{SV} will not increase with the system size. Second, the maximum number of states $2^{N_{SV}}$ is, in practice, significantly reduced because the constraints will prohibit a number of combinations of variable assignments.

Phase 3: Sequence Diagram Annotation

In this phase of our algorithm, we annotate the input SDs in a way similar to that proposed by Whittle and Schumann [21]. The purpose of the annotation process is to enrich the information captured in the SDs by determining the most general conditions that need to hold for the execution of the scenario. For each SD, we create an annotated SD_C for each participating component C. SD_C represents the scenario from C's perspective. To annotate an SD, we make two passes through it. The first pass adds an initial set of annotations that pertain to the individual operation invocations, while the second pass propagates values between adjacent annotations.

To incorporate the information from SDs into the initial MTSs from Phase 2, we need to know which states of the MTSs are traversed when the sequence of operations executes as specified. Therefore, for each component, we enrich the SDs with annotations that describe the necessary conditions on the component's significant variables at particular points of the execution. Each annotation is a vector, and each field in the vector corresponds to a component's significant variable. Variables that must be true are annotated with 'T' and those that must be false are annotated with 'F'. In the case of Cache, for example, each annotation will characterize the necessary values of $requestPending$ and $cached$ at different points of the Web cache system scenarios (Figure 1).

Create the initial set of annotations.

We create the initial set of annotations directly from the component-level constraints. The initial annotations capture the general conditions that have to be satisfied before and after operations are invoked. The annotation before (after) an operation specifies the values of significant variables that have to hold to satisfy the operation's precondition (postcondition). The initial annotations we create are *minimal annotations*.

DEFINITION 10 (MINIMAL ANNOTATION). *For all components C, SDs SD_C, and an operation invocation instances i in SD_C, an annotation $an_{C,i}$ before (after) i is a* minimal annotation *iff $an_{C,i}$ satisfies i's precondition (postcondition), and for all fields $an_{C,i}[v]$ in the annotation vector with a specified truth assignment, modifying the field would violate i's precondition (postcondition).*

By creating only minimal annotations we ensure that (1) we do not create annotations which violate the constraints, and (2) the most general conditions are captured (i.e., adding more undefined fields violates the constraints). Details of the creation of initial annotations can be found in Figure 7. For clarity, we focus on the case when there exists a unique minimal annotation that captures the necessary conditions before (after) the operation invocation. In practice, however, multiple annotations may be needed to capture these conditions (e.g., when a component's state is changed depending on the value of operation parameters). In this case we create multiple copies of the SD with different valid annotations.

Figure 8a depicts the initial set of annotations from Cache's perspective for the Web cache system Scenario 1. The initial annotation before the first invocation of *requestCacheData* in Figure 8a is ⟨F,?⟩ because $requestPending$ must be false to satisfy the *requestCacheData*'s precondition, while $cached$ is left undefined.

Propagate the annotation values.

The initial set of SD annotations captures the conditions that must be satisfied before and after any invocation of individual operations. In an SD however, an operation invocation is preceded and/or followed by other invocations; the surrounding context of an invocation can impose additional conditions that have to hold. Specifically, the annotation after an invocation in the SD should not conflict with the annotation before the next invocation in the

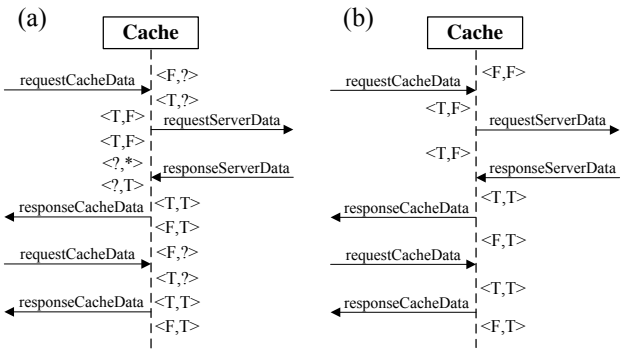

Figure 8: Annotated SD from Cache's perspective (a) after initial annotation, and (b) after propagation. Annotation fields specify values of ⟨requestPending,cached⟩.

```
PROPAGATE(SD SD_C, comp C)
1   boolean changeFlag = true
2   while changeFlag = true
3     changeFlag = false
4     for each invocation i ∈ SD_C
5       for each C''s significant variable v
6         if an_{C,i}[v] ≠ ?  and an'_{C,i}[v] = ?
7           an'_{C,i}[v] = an_{C,i}[v] and changeFlag = true
8         if an_{C,i}[v] = ?  and an'_{C,i}[v] ≠ ?
9           an_{C,i}[v] = an'_{C,i}[v] and changeFlag = true
10    for each adjacent invocation pair (i_k, i_k + 1) ∈ SD_C
11      for each C''s significant variable v
12        if an'_{C,i_k}[v] is assigned and an_{C,i_{k+1}}[v] ∈ {?, ⋆}
13          an_{C,i_{k+1}}[v] = an'_{C,i_k}[v] and changeFlag = true
14        if an'_{C,i_k}[v] ∈ {?, ⋆} and an_{C,i_{k+1}}[v] is assigned
15          an'_{C,i_k}[v] = an_{C,i_{k+1}}[v] and changeFlag = true
16  report unification conflicts and inconsistencies
17  return SD_C
```

Figure 9: The SD value propagation steps.

SD (we assume there are no side effects between invocations). In other words, the two annotation vectors should be *consistent*.

DEFINITION 11 (VECTOR CONSISTENCY). *For all components C and annotation vectors $an1_C$ and $an2_C$ of component C's significant variable assignments, $an1_C$ and $an2_C$ are* consistent *iff for each variable v, $an1_C[v] = an2_C[v]$ whenever both $an1_C[v]$ and $an2_C[v]$ have a defined truth assignment.*

The objective of the propagation is for the already consistent annotations after one invocation and before the next in the SD to be identical. Hence, we propagate values between these adjacent annotations: each annotation field $an1_C[v]$ which is undefined in one annotation is assigned the value of that field $an2_C[v]$ in the other annotation if $an2_C[v]$ is defined. The new value is then also propagated to the undefined significant variable in the other annotation of the same operation if the operation does not modify the value of that variable. We iterate through an SD and apply these propagation steps as long as there are values that can be further propagated. The details of the propagation process are elaborated in Figure 9.

Figure 8b shows the final annotated SD from Cache's perspective. Observe that the annotation before the topmost *requestCacheData* invocation became ⟨F,F⟩, although it was initially ⟨F,?⟩. The propagation of the *cached* value occurred as follows. The initial annotation before *requestServerData* imposes *cached* to be false. This value is propagated to the annotation after *requestCacheData* which initially had *cached* undefined. Since *requestCacheData* does not modify *cached*, the value (false) is further propagated to the annotation before *requestCacheData*. We can infer from the topmost annotation that there are no pending requests and the data entry is not cached before the specified scenario executes.

This phase of our algorithm can also account for the SD loop construct by checking whether the annotations before and after the loop are consistent. This phase also discovers and reports conflicts that appear during the propagation. A discovered conflict implies that the relevant scenario cannot occur in the system as specified. We describe the reasons for and implications of these conflicts in Section 4.1. We also create a system-level annotated SD_{SYS} in the same manner, which is then used to discover potential design flaws (Section 4.2).

Complexity analysis: This phase of our algorithm has the worst-case complexity of $\Theta\left(N_C \times N_{SD} \times L_{SD}^2 \times 2^{N_{SV}}\right)$. This worst-case occurs for extremely complex constraints that include the majority of the significant variables. Since high-level constraints constructed manually by architects tend to be much simpler [1], the factor exponential in N_{SV} will, in practice, actually be polynomial.

Phase 4: Final MTS Generation

In the last phase of our algorithm, we leverage the set of initial MTSs and the set of annotated SDs to construct the set of final component-level MTSs. In the process of MTS refinement, we first determine the MTS states from which a scenario can execute, then traverse the MTS according to the annotated scenario, and convert the traversed potential transitions to required.

Determine the launching state(s) for a scenario.

In our approach, we do not require the scenarios to start executing from the initial components' states. We deduce the set of MTS states from which the scenario execution can start, which we refer to as *SD's launching MTS states*, based on the following definition.

DEFINITION 12 (LAUNCHING MTS STATE). *For all components C, SDs SD_C, and MTSs M_C with the state set S and the truth assignment mapping Map, $s \in S$ is a launching MTS state for SD_C iff $Map(s)$ is consistent with the first annotation in SD_C.*

For example, only the launching state in Cache's initial MTS (Figure 5) for annotated SD SD_{Cache} from Figure 8b is s_0 because $Map(s_0) = \langle$F,F$\rangle$ is identical to the first annotation in SD_{Cache}.

Traverse through the MTS.

For each state in the set of launching MTS states for an SD, we traverse the MTS starting from that state. The first traversed transition $s_1 \xrightarrow{op} s_2$ is labeled with the name of the first invoked operation *op* in the SD, while s_1 is the launching state and s_2 is a state consistent with the annotation after *op* in the SD. If the traversed transition t is a potential transition, we refine the MTS by making t required, provided that conditions discussed in the next step hold. We then perform the same step from t's destination state for the next invocation in the scenario, and repeat this process for all the invocations in the SD. In case of Cache, the first transition we traverse is $s_0 \xrightarrow{requestCacheData}_p s_2$. Figure 10 provides a detailed description of the MTS traversal.

Refine the MTS with required behavior from SDs.

Refining a traversed potential transition t by simply modifying it to required would make the resulting MTS overspecified. For example, imagine we traverse Cache's initial MTS from Figure 5 over the transition $s_2 \xrightarrow{requestServerData}_p s_2$ for some SD. If this transition was modified to required, we would introduce a required self-loop in s_2 on *requestServerData*. This would make the MTS overspecified since it would now impose that subsequent invocations of *requestServerData* must be supported, although the SD requires that only one such invocation is necessarily supported.

```
GENERATEFINALMTS(MTS M_C, SD SD_C)
1   MTS M = M_C
2   {stateSet S' ⊆ M_C.S where s ∈ S'
3   satisfy annotation before SD_C.firstOp}
4   for each op ∈ SD_C.orderedOperations
5     stateSet S_next = ∅
6     for each s ∈ S'
7       for each t: s -op-> s_2 in M
8       where s_2 satisfies the annotation after op
9         MTS N = M
10        if t is a required transition
11          S_next.add(s_2)
12        if t is a potential transition
13          if ∃t_2: (s_3 -op->_r s_2) in M
14            set t required in N
15            S_next.add(s_2)
16          else (s', s'') = REFINE(M, N, s, s_2, op)
17            set s' -op->_r s'' in N
18            S_next.add(s_2)
19        M = N
20    S' = S_next
21  return M

REFINE(MTS M, MTS N, st s, st s_2, oper op_curr)
1   {refine s_2 in N into s'_2 and s''_2
2   where Map(s'_2) = Map(s''_2) = Map(s_2)}
3   if s = s_2 in M
4     s = s'_2 in N
5   for each t_2: s_3 -op'-> s_2 in M and s_2 ≠ s_3
6     create t'_2: s_3 -op'-> s'_2 in N
7   for each t_2: s_2 -op'-> s_3 in M and s_2 ≠ s_3
8     create t'_2: s'_2 -op'-> s_3 in N and t''_2: s''_2 -op'-> s_3 in N
9   for each t_2: s_2 -op'-> s_2 in M
10    create t'_2: s'_2 -op'-> s'_2 in N and create t''_2: s''_2 -op'-> s'_2 in N
11  for each t_2: s_3 -op_curr-> s'_2 in N
12    create t'_2: s_3 -op_curr-> s''_2 in N
13  return (s, s''_2)
```

Figure 10: Final MTS generation phase.

To address this issue, we refine the destination state s_2 of a traversed transition t labeled with *op* into two new states s'_2 and s''_2. State s''_2 is the new destination of all of s_2's incoming transitions labeled with *op*, while s'_2 is the new destination state for s_2's remaining incoming transitions. Transition t (terminating at s''_2)is then modified into a required transition and the next step in the MTS traversal is performed from s''_2. REFINE in Figure 10 details these steps. Finally, after iterating over all of the annotated SDs, the final MTS has as much of its potential behavior refined to required as can be determined from the provided SDs, while remaining a strong refinement of the initial MTS as we prove below.

Figure 11 shows the first two steps in the MTS refinement for Cache and the final MTS that is obtained after stepping through the whole SD_{Cache} (recall Figure 8b). The bolded parts in the first three MTSs show the traversed transition, its source and destination state, as well as the transition that was refined in the previous step. The enumerations in Cache's final MTS depict how Cache supports Scenario 1 from Figure 1 through required transitions.

In the first step of Cache's MTS refinement, we traverse the transition $t\colon s_0 \overset{requestCacheData}{\rightarrow_p} s_2$. We refine the destination state s_2 to s''_2, which has all of s_2's incoming transitions defined on *requestCacheData*, and state s'_2, which has s_2's incoming transitions defined on *requestServerData*. We also modify t to become a required transition. The next step in Figure 11 shows the result obtained after traversing the transition $s''_2 \overset{requestServerData}{\rightarrow_p} s'_2$ corresponding to the invocation of *requestServerData* in the SD. The final MTS satisfies all constraints from Figure 1 and realizes the sequence described in Scenario 1. The final MTS also captures behavior that has yet to be decided, such as whether Cache can repeatedly invoke *requestServerData*.

Complexity analysis: Similarly to Phase 3, the worst-case complexity of Phase 4 is $\Theta\left(N_C \times N_{SD} \times L_{SD}^2 \times 2^{N_{SV}}\right)$. By using appropriate data structures, we can reduce this complexity to $\Theta\left(2^{N_{SV}} \times N_{COP} \times N_C\right)$. As discussed earlier, the constraints' nature will render the practical complexity polynomial in N_{SV}.

Refinement Proof

In this section we prove the strong refinement relation between an initial MTS and the final MTS generated by our algorithm.

THEOREM 1. *The initial MTS and the final MTS produced by the presented algorithm share the strong refinement relation.*

PROOF. (by induction)

Base case: By definition of strong refinement (Definition 3), the initial MTS is its own strong refinement.

Inductive Hypothesis: Given an MTS M, after a single iteration of the algorithm from Figure 10 on a transition t (the loop in lines 7–19 of GENERATEFINALMTS), the produced MTS N is a strong refinement of M. Because the strong refinement relation is transitive, proving the inductive hypothesis implies that the final MTS is a strong refinement of the initial MTS.

We start by constructing a relation between states in M and N. This relation depends on the transition t. If t is a *required* transition (lines 10–11 of GENERATEFINALMTS), N is identical to M, which preserves the strong refinement relation. If t is a *potential* transition, $s \overset{op}{\rightarrow_p} s_2$, and there exists a required incoming transitions into s_2 labeled with *op* (lines 13–15 of GENERATEFINALMTS), then N has a single *potential* transition modified to a *required* transition, and thus N is a strong refinement of M.

The final MTS N may, by construction (ensured by REFINE), have only two distinct types of states: type 1 states have only incoming potential transitions, and type 2 states have at least one incoming required transition and all the incoming transitions are labeled with the same operation. Dealing with the traversed transitions with a destination type 1 states is already discussed above.

REFINE and lines 16–18 of GENERATEFINALMTS take care of traversed transitions $t\colon s \overset{op}{\rightarrow_p} s_2$ that reach type 2 states and ensure that $R = \{(s_i, s_i) | s_i \neq s_2\} \cup \{(s_2, s'_2), (s_2, s''_2)\}$, which we will show, is a strong refinement relation. For all the MTS states that do not have a transition to s_2, no modifications are made, so R is a strong refinement for those states. For each state s_j, $s_j \neq s_2$, with outgoing transition to s_2, the first strong refinement condition (see Definition 3) is directly satisfied since type 2 states have no incoming required transition. The second condition of strong refinement is satisfied because:

1. For each transition $(s_j \overset{l}{\rightarrow_p} s'_2)$ in N, transition $(s_j \overset{l}{\rightarrow_p} s_2)$ existed in M (ensured by lines 5–6[2]).
2. For each transition $(s_j \overset{l}{\rightarrow_p} s''_2)$ in N, transition $(s_j \overset{l}{\rightarrow_p} s_2)$ existed in the M (ensured by lines 13–14).

Finally, we show that $(s_2, s'_2) \in R$ and $(s_2, s''_2) \in R$ satisfy the strong refinement conditions. We do so by enumerating over all possible transitions to and from those states:

1. For each transition $(s_2 \overset{l}{\rightarrow_r} s_j)$ in M, N has required transitions $(s'_2 \overset{l}{\rightarrow_r} s_j)$ and $(s''_2 \overset{l}{\rightarrow_r} s_j)$ (ensured by lines 7–9). This statement satisfies the first condition of the strong refinement definition.
2. No transition $(s'_2 \overset{l}{\rightarrow_p} s_j)$ in N, such that $s_j \neq s'_2$ and $s_j \neq s''_2$, violates the second strong refinement condition as transition $(s_2 \overset{l}{\rightarrow_p} s_j)$ existed in M (ensured by line 8).

[2]Unless otherwise specified, line numbers refer REFINE.

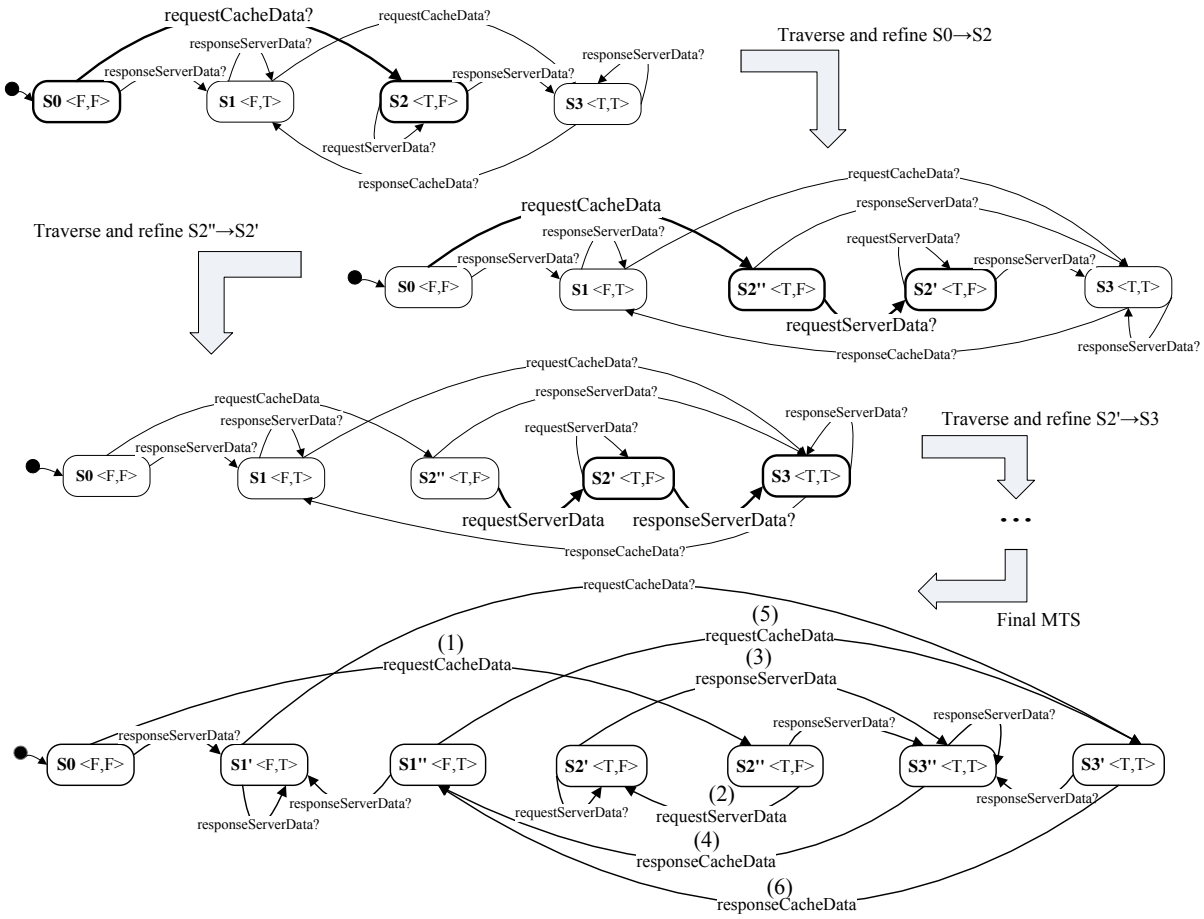

Figure 11: From the initial to the final MTS for cache.

3. No transition $(s_2'' \xrightarrow{l}_p s_j)$ in N, such that $s_j \neq s_2'$ and $s_j \neq s_2''$, violates the second strong refinement condition as transition $(s_2 \xrightarrow{l}_p s_j)$ existed in M (ensured by line 9).
4. No transition of either the form $(s_2' \xrightarrow{l}_p s_2')$ or $(s_2'' \xrightarrow{l}_p s_2')$ in N violates the second strong refinement condition as transition $(s_2 \xrightarrow{l}_p s_2)$ existed in M (ensured by lines 10–12).
5. No transition of either the form $(s_2' \xrightarrow{l}_p s_2'')$ or $(s_2'' \xrightarrow{l}_p s_2'')$ in N violates the second strong refinement condition as transition $(s_2 \xrightarrow{l}_p s_2)$ existed in M (lines 10–14).

Therefore, no transitions in N violate the strong refinement condition, and thus N is a strong refinement of M. □

4. DISCOVERING DESIGN FLAWS

In this section, we further evaluate our algorithm. We discuss how the algorithm aids the discovery of potential design flaws, which, to our knowledge, are overlooked by the existing synthesis approaches. First, the SD annotation phase of our algorithm discovers all scenarios that cannot execute as specified. Second, analysis of the annotations on the component-level and system-level SDs created by our algorithm can suggest subtle design flaws that result from inconsistencies between component states and the expected system state. Third, analysis of the generated MTSs of different components can suggest likely design flaws resulting from overly restrictive or overly permissive constraints. In this section, we explore the origins and implications of each of the three discrepancies. We also propose solutions an architect may apply depending on the nature of the system under development.

To support the exploration of the design flaws and the application of our technique on a number of examples, we have developed a prototype tool, MTSGen [15]. MTSGen currently takes a system's specification in terms of scenarios and properties defined on Boolean variables, and automatically constructs the component-level MTSs according to our algorithm. To assess the scalability of our technique, we have used MTSGen to automatically generate component MTSs from specifications comprising up to 50 components, 300 system operations, 200 domain variables, and 200 scenarios. The average duration of the synthesis for the largest resulting specifications was, on average, 36 seconds on a mid-range Windows PC, while the resulting models had approximately 60 states. This is consistent with the specification and resulting model sizes we expect to see in real-world systems.

4.1 Scenario Cannot Execute as Specified

By modeling the behavior of the system in two different and complementary ways, namely, via scenarios and properties, the architect is forced to truly understand the system specifications and the behavior implied by those specifications. In the propagation steps of SD annotation (Phase 3 of our algorithm from the previous section), we discover discrepancies between the input scenarios (SDs) and properties (constraints). Discrepancies arise when a scenario is supposed to exhibit behavior that is prohibited by some constraint. The discrepancy is discovered in the following two cases: (1) when the annotations after one and before the next

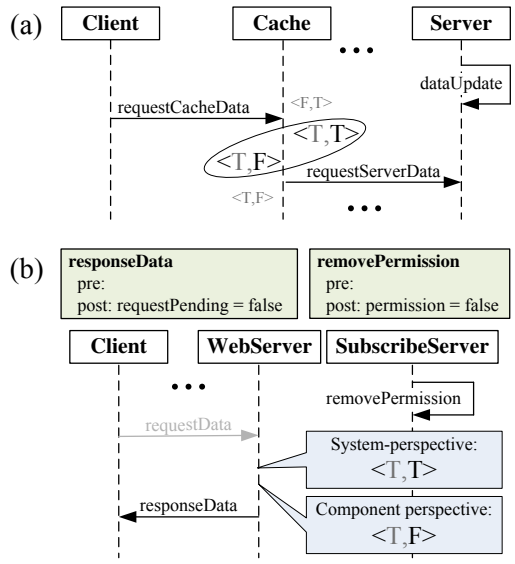

Figure 12: Example scenarios with specification discrepancies.

invocation in the SD are not consistent, and (2) when an operation does not modify a particular significant variable v, but the annotation fields corresponding to v before and after the invocation are defined and have different values.

For example, in the specification of the Web cache system (Figure 1), Scenario 2 contains a discrepancy. The discrepancy shown in Figure 12a is unveiled during the SD annotation from Cache's perspective: the value of $cached$ in the annotation after *requestCacheData* is false, while it is true in the annotation before *requestServerData*. According to the scenario, Cache should request Server data after the invocation of *dataUpdate*. However, Cache is not aware of that invocation and considers the data cached. Therefore, *requestServerData* invocation would not occur as $cached =$ true conflicts with the precondition of *requestServerData*.

A conflict of this type has multiple possible causes and solutions. For instance, one or more constraints may be overly restrictive (i.e., the scenario is valid, but a constraint prevents it). In this case, the architect should consider relaxing the constraint that prevents the scenario's execution. More importantly, the architect should investigate the reasons why the constraint is not required to hold for the particular scenario. Are there special cases that require application of a different constraint set? Is the system in a different operating mode in which some constraints are irrelevant? For example, the problem in Figure 12a may be resolved by relaxing the constraint on *requestServerData*. However, this would require additional analysis to ensure that undesired behavior is not introduced.

Another cause of this type of discrepancy is that the scenario is misspecified and one or more constraints correctly prevents its execution. In this case, the architect should either correct the scenario or remove it. This problem can result if the scenario is lacking a required invocation, performs prohibited invocations, or invocations occur out of order. The underlying cause of the conflict may be a simple oversight during the scenario's construction, or it may indicate a larger issue that requires design modifications. For example, a component may be lacking execution information because it does not have access to the required resources. The problem from Figure 12a may be resolved by adding to Cache a new provided operation *dataChanged* which changes $cached$ to false. Server would need to provide a notification of every *dataUpdate* invocation by invoking *dataChanged*, and the discrepancy would be removed.

4.2 System And Component Views Differ

There are cases when a scenario can execute according to both the component- and system-level perspectives, but undesired behavior can still occur due to internal component states that differ from the expected system state. For example, consider the Web server system from Figure 12b, which resembles the Web cache system. In the Web server system, WebServer provides partial data to an unsubscribed Client, and full data if Client is subscribed. In the depicted scenario, SubscribeServer, which manages the subscriptions, removes Client's subscription thus making a domain variable $permission$ false. Because *responseData* does not have any preconditions, the scenario executes correctly. However, comparison of annotations in $SD_{WebServer}$ and SD_{SYS} discovers that fields corresponding to $permission$ differ (highlighted in the diagram), which, in this particular case, unveils undesired behavior.

These types of issues can thus be uncovered by comparing each component's annotated SD, which captures the component's internal states, with the system-wide annotated SD, which captures the expected and desired states at different points of the scenario execution. Intuitively, the comparison of component-level and system-wide annotations provides an automatic determination of which components are not "in sync" with the expected system state. These state inconsistencies can result from either of the following causes: (1) a scenario allows a domain variable to be legally modified via some invocation, but all interested components are not notified, hence their states become inconsistent with the system state, (2) a scenario allows a variable to be modified in a manner that is incompatible with the system-wide scenario; as a consequence, a component with an inconsistent state may perform an invocation that also moves the callee to an inconsistent state.

If such state inconsistencies are discovered, the architect should decide whether the specified behavior is indeed legal behavior. If not, the most common strategy for addressing these problems is to add a method invocation to the scenario that notifies all the relevant components of the new variable value. Ultimately, the discovery of this issue can lead to substantial design modifications. For example, if a variable is discovered whose value is of interest to many components, the architect may elect to employ a publish-subscribe architecture to distribute updates.

4.3 Component-Level MTSs Differ

The final output of our algorithm is a set of component-level MTSs. The reachable states in a component MTS represent the valid combinations of the significant state variables for that component. We can perform comparisons of the different components' MTS state sets to enumerate the valid value combinations for significant variables that are not in common to the components. In the Web cache system, for example, if the initial values of domain variable $cached$ is true, then Cache's MTS would only have states in which $cached =$ true, while Server's MTS would have states where $cached =$ true and states where $cached =$ false.

Such a discrepancy may (though it need not) indicate a design flaw and should be further analyzed. There are two possible causes of the discrepancy. First, the system may be underconstrained so that certain undesired behaviors are not prevented and a component can end up in an illegal state. In this case, the architect should modify existing constraints to make them more restrictive or add new constraints to explicitly disallow the behaviors leading to the illegal state. Second, component may be overconstrained and, as a result, is unable to reach a desirable state. To address this issue, the architect should relax the constraints that apply to the component or introduce new operations that lead to the desired state. As noted before, however, caution must be exercised when relaxing constraints

to ensure that invalid behavior is not introduced, and the reason that the constraint does not apply should be understood fully.

5. RELATED WORK

There exist a number of approaches for synthesis of system- and component-level behavior models from system specifications. As we noted earlier, the existing approaches for component-level model synthesis produce final (as opposed to partial) models. Whittle and Schumann [21] proposed an algorithm for generating component statecharts from scenarios and properties. Their algorithm works on similar inputs as our algorithm, but without considering the behavior that is neither prohibited nor required by the specifications. Uchitel et al. [18] demonstrate the importance of considering the specifications' partiality during architectural refinement.

Mäkinen and Systä [13] developed a semi-automated approach for using architect guidance to synthesize component statecharts from SDs. Damas et al. [3] propose inductively inferring a system-level LTS and subsequently decomposing it into component-level LTSs from scenarios interactively provided by the user. A later extension of this approach reduces the number of questions to the user [4] by incorporating FLTL properties [8]. However, these techniques can synthesize overspecified models.

Uchitel et al. [19] put forward a technique for component-level LTS model synthesis from MSCs [10] and discovering implied scenarios. The resulting LTS models are constructed from LTS models of individual scenarios, which are composed according to an additional high-level MSC. Our algorithm does not require the architect to specify explicit relations between scenarios. The approach by Harel et al. [9] synthesizes statecharts from LSCs [5]. In their approach, additional events that synchronize the states of different components augment the statecharts.

Recent relevant approaches construct LTSs based on pre- and postcondition specifications [2, 6]. De Caso et al. [6] focus on generating more abstract models than we do, in order to support validation of the specifications. Their tool Contractor accepts a rich set of variable types, which we plan to offer in MTSGen in the future. De Caso's approach, however, does not consider other types of specifications. Alarjeh et al.'s technique [2] facilitates refinement of pre- and postconditions based on system goals and scenarios. These two approaches natively complement our approach by supporting validation and refinement of properties at the system level.

The work we present in this paper addresses the problem of inherent partiality of early system specifications. The main distinction between our work and the work done in [17, 20] is that we generate component-level, as opposed to the system-level, MTS models. Work by Uchitel et al. [20] generates system-level MTSs from scenarios and property specifications while imposing each scenario to start from the initial MTS state. Conversely, we determine the starting state for a scenario based on the SD annotations. Sibay et al. [17] present a novel scenario specification formalism, epLSC, and demonstrate synthesis of system-level MTSs from epLSCs.

In general, none of the above approaches accounts for the types of design flaws our algorithm helps to discover (Section 4).

6. CONCLUSIONS

In this paper, we proposed a novel technique for synthesizing partial component-level behavior models from high-level system specifications. Generating partial component-level models, as opposed to system-level models, introduces more rigor into the requirements elicitation and architectural refinement processes and aids the discovery of several classes of potential problems. We described our algorithm in detail, analyzed its complexity, and demonstrated the types of specification discrepancies we can discover. In our on-going work, we are evaluating the usefulness of the discovered discrepancies, and are exploring ways of using the component MTSs for requirements elicitation and off-the-shelf component selection. We are also working on extending our algorithm to encompass additional specifications, including information about system goals, architectural styles, and negative scenarios.

Acknowledgments

This work is supported by the National Science Foundation under Grant numbers ITR-0312780, CSR-0720612, and SRC-0820170.

7. REFERENCES

[1] J. Ackermann and K. Turowski. A library of OCL specification patterns for behavioral specification of software components. In *Proc. of CAiSE*, 2006.

[2] D. Alarjeh et al. Learning operational requirements from goal models. In *Proc. of ICSE*, 2009.

[3] C. Damas et al. Generating annotated behavior models from end-user scenarios. *IEEE TSE*, 31(12), 2005.

[4] C. Damas et al. Scenarios, goals, and state machines: a win-win partnership for model synthesis. In *Proc. of FSE*, 2006.

[5] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Form. Meth. Syst. Des.*, 19(1), 2001.

[6] G. de Caso et al. Validation of contracts using enabledness preserving finite state abstractions. In *Proc. of ICSE*, 2009.

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

[8] D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *Proc. of FSE*, 2003.

[9] D. Harel et al. Synthesis revisited: Generating statechart models from scenario-based requirements. *Form. Meth. in Soft. and Sys. Modeling*, 3393, 2005.

[10] ITU. Message sequence charts, 2000.

[11] I. Krka, G. Edwards, Y. Brun, and N. Medvidovic. From system specification to component behavioral models. In *Proc. ICSE NIER*, 2009.

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

[13] E. Mäkinen et al. MAS–an interactive synthesizer to support behavioral modeling in UML. In *Proc. of ICSE*, 2001.

[14] R. Milner. *Communication and concurrency*. Prentice-Hall, 1989.

[15] MTSGen. `http://www-scf.usc.edu/~krka/MTSGen.zip`.

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

[17] G. Sibay et al. Existential live sequence charts revisited. In *Proc. of ICSE*, 2008.

[18] S. Uchitel et al. Behaviour model elaboration using partial labelled transition systems. In *Proc. of ESEC/FSE*, 2003.

[19] S. Uchitel et al. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM TOSEM*, 13(1), 2004.

[20] S. Uchitel et al. Behaviour model synthesis from properties and scenarios. In *Proc. of ICSE*, 2007.

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