Software Behavior Synthesis

Hesham Shokry
Lero – The Irish Software Engineering Research Centre
University of Limerick, Ireland

Mike Hinchey
Lero – The Irish Software Engineering Research Centre
University of Limerick, Ireland

26th April 2010

Contact

Address ..... Lero
International Science Centre
University of Limerick
Ireland
Phone ..... +353 61 233799
Fax ..... +353 61 213036
E-Mail ..... info@lero.ie
Website ..... http://www.lero.ie/

This work is partially supported by Science Foundation Ireland
under grant no. 03/CE2/I303-1

THE IRISH SOFTWARE ENGINEERING RESEARCH CENTRE

Software Behavior Synthesis (Draft)

Hesham Shokry
Mike Hinchey

Lero – The Irish Software Engineering Research Centre
University of Limerick, Ireland

26 April 2010
[Version 1.0]

This work is partially supported by Science Foundation Ireland under grant no. 03/CE2/I303-1.

Lero Technical Report Lero-TR-2010-01

ABSTRACT
Early system requirements are often captured by declarative and property-based artifacts, such as scenarios and goals. While such artifacts are intuitive and useful, they are partial and typically lack an overarching structure to allow systematic elaboration of the partial behaviors they denote. We propose a structuring approach appropriate for scoping different partial behaviors, focusing on scenario-based behavior specifications. The approach is based on Parnas' notions of 'modes' and 'mode-classes', where a mode is a set of states that satisfy some predicate, and a mode-class is a collection of disjoint modes that partitions the system's state-space so that each state belongs to exactly one mode. There may be several mode-classes, in which case every state belongs to exactly one mode from each mode-class. We structure a scenario by partitioning its observed states into modes, allowing elaboration of the scenario's parts independently without losing the overall system view. Having every scenario partitioned via a suitable mode-class, we merge the mode-classes constructively to build a single behavioral model of the system. We argue that our approach facilitates early refinement and an improved coverage of requirements, as well as improved generation of system models from partial behaviors. We provide a sound formal model of modes, based on which we detail a novel technique to synthesize a prototype of system behavior, given a set of scenarios and corresponding mode-classes specifications as input.

Keywords
Mode-based design, Behavioral-models synthesis, State-space partitioning.

1. INTRODUCTION

At early stage of development, designers often have little or vague information regarding requirements and the given information is subject to frequent changes. This can result in too much iteration between design and requirements specification activities. A general solution for overcoming these problems is prototyping system behavior to be used for early reasoning and elaboration. However, vagueness about behavior space and varieties of sources of requirements results in specifications that are unstructured and partial. Partial specifications are commonly captured via intuitive artifacts such as *goals* [1],[2] and *properties* [3], also known as *declarative specifications*. Another type of partial specification, of particular interest to this report, is scenario-based specifications [4, 5].

The problem with partial specifications is that designers will be unable to (1) ensure space-coverage of prospective system behavior, nor (2) reason about system-level properties that cross boundaries of the individual specifications fragments (such as scenarios). Both issues require a system model that spans the behavior space and that can successfully glue the isolated behaviors together into one integrated behavioral model.

Several approaches, however, have attempted a (semi-) automatic synthesis of system-level (e.g. [6]) and component-level models (e.g. [7, 8]) from partial specifications. A common characteristic of these approaches is that they address the *symptom* of the problem: partiality, instead of its cause: the lack of a proper overarching structure of the system behavior. More specifically, most approaches rely on unstructured scenarios

specifications without providing *contextual* information defined for individual snippets of the specifications so as to organize them and relate them in the system's state-space.

A lack of such contextual information leaves system designers with no option other than ad hoc specifications, and no criteria to decide when (i.e. at which state) to start a scenario and when to stop it. This leads to implicit traversing of inter-contexts, which in turn impedes the opportunities to elaborate the individual behaviors and discover more requirements. Synthesis techniques that rely on unstructured specifications pose problems such as:

(1) Restrictive assumptions; e.g., scenarios should start at an initial state [6] which is not necessarily the case for all possible scenarios.

(2) Problems arise when merging the independently-generated behaviors from the partial specifications. For example, generation of nondeterministic transitions [8] and unavailability of a *common refinement* [6].

Our hypothesis is that the provision of system contextual information allows us to scope and structure partial specifications, and enables improved coverage of requirements. This in turn will expose more opportunities for requirements elaboration and, when gaps are discovered, new requirements are elicited. In this report, we propose an approach for structuring the partial specifications by specifying them under pre-defined scopes or *contexts*. A *system context* is a (compound) condition of the system, defined as a predicate over system variables. The system state space is structured into a set of contexts, where different pieces of specifications (e.g., different scenarios) are defined exclusively within the scope of these contexts. Our structuring approach is based on a disciplined partitioning of system state space using Parnas' notion of *mode-classes* as introduced in A-7E project [9, 10]. A mode-class completely partitions the state-space into disjoint clusters or *modes*. Every mode is simply defined as a predicate over system variables.

It worth noting that the term 'mode' is defined and used for two fundamentally different purposes in the computing literature: *modal logic* [11] and *hybrid systems* [12]. The concept we use here originates in the latter.

The next section of this report provides a brief background on behavioral modeling approaches and synthesis from partial specifications (Section 2.1) and then we describe an example motivating our approach (Section 2.2). Section 3 provides definitions and basic formal basis of the *modes* and *mode-class* notions. Based on this formal model of modes, we describe in Section 4 a synthesis technique that accepts structured specifications in the form of scenarios organized with mode-classes and, as output, it generates an integrated state-based model of the system. Related work is discussed in Section 5.

2. BACKGROUND AND MOTIVATION

In this section, we, firstly, provide some background on scenario-based specifications and different behavioral modeling approaches. Secondly, we motivate our approach by an example system.

2.1 Background

2.1.1 Scenario-Based Partial Specifications

There are a variety of ways of describing scenarios. This ranges from informal UML sequence diagrams [4], possibly annotated with OCL pre/post constraints, to a more formal Message Sequence Charts, MSC, [5]. An overview of the spectrum of different dialects of scenario-based specifications can be found in [13]. A *higher-level* form of scenarios such as higher-level MSC (hMSC) [14] and interaction overview diagrams (IOD) [4], provide a flowchart-like composition of lower-level scenarios, which is still a form of scenario involving some control-flow constructs. To focus on the ideas here, we use basic sequence diagrams, where operations are annotated with pre/post conditions in the form of valuations of the vector of system variables.

Despite the debate [15],[16],[17] about what a *scenario* really is, a general accepted interpretation is a sequence of interactions steps between the computer system and the outside environment. A scenario is *partially* describing the computer system behavior because it specifies its reactions to the environment's stimuli *as far as* the scenario is concerned. So, the scenario specifications *completely* specify the computer system behavior only if they specify *all possible* environment stimuli and possible combinations. In practice, however, such complete requirements are not readily available, particularly at early development stage. A fundamental reason is that, typically, scenarios are provided by different stakeholders with different viewpoints and needs [18].

From an automata-based viewpoint, each step in a scenario (action on, or reaction from, the computer system) is perceived as *the progress of the system's automaton* in the sense that each step *modifies* one or more domain variables. In a scenario involving a computer system represented as one component, the successive scenario steps *induce* an automaton[1] representing a black-box (or interface) behavior of the computer system with respect to its environment. However, in the presence of structural decomposition of the computer system itself, architects are also interested in scenarios describing interactions between the system's internal components. In the latter case, the scenario steps induce an automaton representing a clear-box internal behavior describing how the system *implements* its reactions in terms of cooperation between its internal components (note that the induced states involve domain variables plus system-internal variables).

This observation, amongst others, has motivated several approaches (cf. [6],[7],[8]) to exploit scenario-based specifications for the purpose of synthesizing an integrated automata-based behavioral model of the system, given a set of scenarios (or other forms of partial specifications) as input. A common denominator of these approaches is the focus on issues related to partiality of specifications, such as detecting negative-scenarios , implied scenarios [19], and merging behaviors that are independently compiled from separate scenarios [20]. Synthesis processes developed in these approaches have raised some issues. For example, constraining the scenarios by assumptions such as scenario must start from the system's initial-state [6]. The partiality-related issues have distracted the synthesis techniques to solve these issues than to focus on the other challenges of the synthesis process itself. So, in our research work we pay attention to minimize partiality in the specifications before the synthesis.

[1] The reader should remember that this automaton is (typically) just a navigation path in the complete system's observable behavior.

The ultimate way to reducing partiality in specifications is to maximize the opportunities of elaboration and elicitations of requirements, ideally before the synthesis phase. We propose in this report an approach attempting to maximize such opportunities, making use of the two most fundamental principles of software engineering: abstraction and separation of concerns. We partition the state space completely into a manageable set of disjoint sub-spaces each of which represents a context of the system, and then we specify scenarios within those contexts. The inter-contexts transitions are represent as transitions between the scenarios. We use the notion of mode-classes, early introduced by Parnas [9, 21], to model the partitioning of state-space. *Modes* provide the necessary abstraction of the state space, which is necessary at early stages of developments where incomplete information about the individual system states is known. The exclusive (i.e. disjoint) contexts provide the separation of concerns between different scenarios and hence increase the opportunity for their elaboration.

2.1.2 Behavior Abstraction

Compared to the established structural modeling frameworks, such as the popular object-orientation [4] and the original work of modular design [22], there is a limited support for behavior modeling and composition. Typically, system behavior modeling follows structural decompositions. For example, it is not uncommon for designers to specify the automaton of each individual component and then use automata-composition techniques to construct a system behavioral model. A Scenario between the system's components is another example of a behavioral model that follows structural decomposition—interactions between components are based on their relationships in the structural model (e.g., require/provide interfaces of each component).

Behavior formalisms such as process algebras and communicating automata are mainly focused on issues related to *parallelism* among a set of interacting processes which are serving as placeholders for components. On the other hand, hierarchical state-based formalisms do not provide appropriate abstractions in sense of the Dijkstra's sound definition quoted in [23]: an abstraction "is one thing that represents several real things equally well".

From an automata-based viewpoint, an abstraction hides unimportant details about state information and shows those details of interest in the abstract model. Models based on the concept of hierarchical states [24], [25] do not abstract states information, but rather they factor-out state information and show them across hierarchical levels assuming that the system could exist in several states at a time. Systems Engineering [26] has stressed that any system is in exactly one state at a time, and we believe that software is not an exception. So, promoting the concept of *several states at a time* actually jeopardizes the fidelity of the model as well as widening the (already existing) gap between system engineers and software engineers.

For these reasons, we propose to use the idea of modes and mode-classes, as introduced in the A-7E aircraft project [9, 10] to formulate an appropriate behavioral modeling framework. We use this framework to support structuring and elaboration of requirements specified in partial forms.

2.2 Motivating Example

Engineered Safety Feature Actuation System, ESFAS [27], is a popular example for illustrating the synthesis of behavioral models from partial specifications (cf. [1]). We use the ESFAS system (with slight modifications) as a running example to illustrate concepts presented in this report where necessary.

The ESFAS component is a computer system, part of a power plant, intended to mitigate damage to the plant on occurrence of faults. ESFAS receives signals from different sensors[2] and checks if the signal level has reached predetermined set-points, in which case ESFAS sends a safety notification to the SafetyHandler component which deals with the accident. The scenario in Figure 1 shows a sample interaction between ESFAS and other environment components.

We identified the following system variables to define the ESFAS state at any point in time:

- the pressure variable *pr*, is *low* or *norm* (i. e., normal value) or *perm* (i. e., permitted) where *low*<*norm*<*perm*;
- the safety signal variable *ss*, is *active* , *inactive*;
- the safety blocking push-button *sb*, is *on* or *off*;

The ESFAS component has the following requirements:

R1. ESFAS activates the safety signal (*ss*=*active*) when *pr* is *low*, and deactivates it (*ss*=*inactive*) when *pr* rises to *norm* level;

R2. ESFAS must not activate the *ss* if the *sb* button is *on*. This enables a human Operator to block the safety so as to prevent unneeded activation during start-up (before ESFAS itself initializes) or cool-down phases. The Operator should reset the button back (*sb*=*off*), or otherwise the ESFAS reset it automatically after *timeout* duration since it has been set *on*.

R3. The Operator can block the safety activation only when *pr*=*low*;

R4. The Operator can not block the *ss* while it is already *active*.

Fig. 1 shows a scenario for the plant. There are two behaviors that are mixed in this scenario:

- the behaviors in **R3** and **R4** are modeled by the messages **M6** and **M9**.
- the behaviors in **R1** and **R2** are modeled by the messages **M1**, **M2**, **M7** and **M8**.

The ESFAS component is required to perform the following:

[2] We consider here only the Pressurizer sensor. Moreover, in the real ESFAS system, a signal is acquired by voting among 3 or 4 redundant sensors channels. This voting logic shown in [26] is omitted here for reasons of brevity.

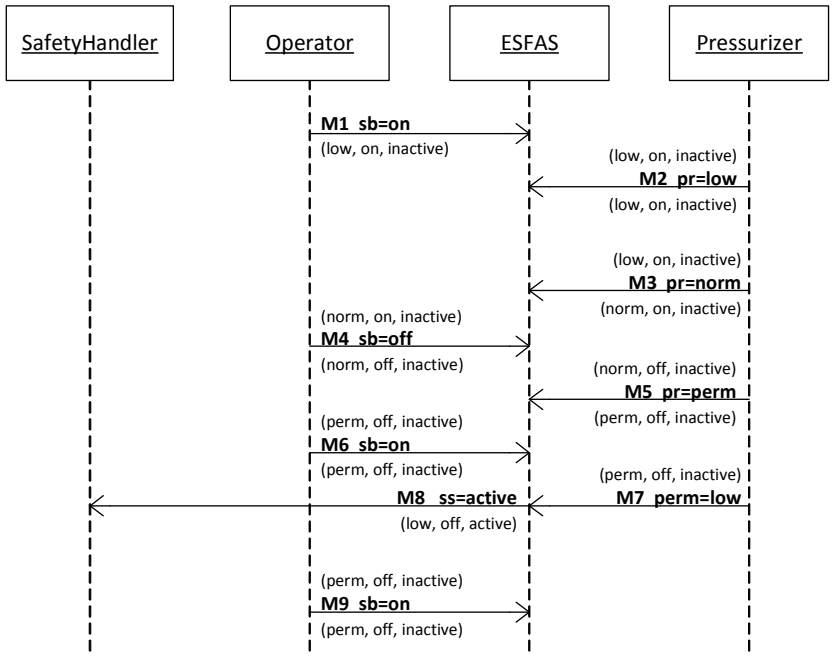

Figure 1: Normal-startup scenario: the Operator blocks the safety signal activation while the system is starting up.

Moreover, ESFAS has the following constraints:

- The Operator can block the safety activation only when **P**=PERM;
- The Operator cannot block the safety activation while it is already active.

These two behaviors are independent but the distinction between them is hidden in the scenario. This is particularly apparent when the scenario is transformed into state-machines, where it would then be difficult to identify which state belongs to which context. The main reason for mixing those behaviors is the lack of systematic techniques to establish contexts in which the scenarios start and stop. In practice, designers insert some interactions before and after the core behavior they wish to show in a scenario, in an attempt to initialize a context for it. Moreover, some approaches assume that a scenario must start at the initial state [6], which is not necessary for every scenario. The main reason for that is the lack of systematic techniques to establish different contexts within which the scenarios are specified.

In summary, the ESFAS example illustrates how the different contexts of the system can be mixed in the same scenario and this impedes elaboration and full coverage of requirements. The absence of a proper overarching framework to structure those contexts, and the scenarios executing within them, allows the specifications to be more partial and hides potential gaps in the state space. In the rest of this report, we present a novel framework to structure and organize these contexts and facilitate an improved synthesis of automata models from scenarios.

3. MODAL BEHAVIOR: FOUNDATION

To manage contextual information, we use the concept of *mode* as an abstraction of states to scope out a specific context. Several system modes are organized via a *mode-class*. Informally, a mode-class is a collection of modes that *completely* partitions the

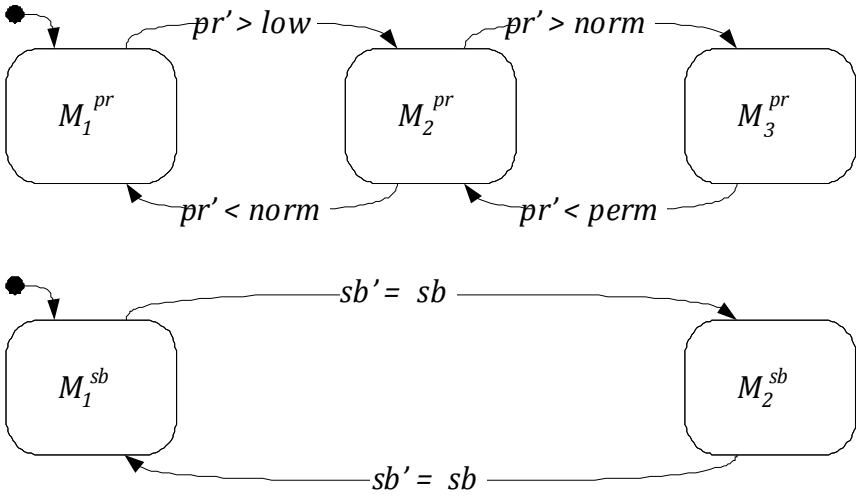

Figure 2: ESFAS mode-classes: (a) Pressure MC^{pr}, (b) Safety Blocking MC^{sb}.

possible system states set into disjoint subsets. Each mode defines a context in which certain behavior(s) can be specified. The same system can be seen as behaving across several mode-classes such that the system exists in exactly one mode from each mode-class. More formally, this means that a system state must belong to exactly one mode from each mode-class. Every mode-class partitions the (same) system from a different dimension or viewpoint. This implies that, at any point in time, *the system is in several modes (each from a specified mode-class) but it is in one-and-only-one state.*

Consider that the state-space of ESFAS is partitioned by the two mode-classes MC^{pr} and MC^{sb}, where MC^{pr} is composed of modes M_1^{pr}, M_2^{pr}, M_3^{pr}, while MC^{sb} is composed of modes M_1^{sb} and M_2^{sb}. A graphical view of these mode-classes is shown in Figure 2, where a mode-class is depicted as a transition-system with an initial mode. Each mode in a mode-class is characterized by a predicate. For example: $M_1^{pr} \Rightarrow (pr \leq low)$, $M_2^{pr} \Rightarrow (pr > low \wedge pr \leq norm)$, $M_3^{pr} \Rightarrow [pr \geq perm]$. Intuitively, a mode is the endurance of the system operation/execution over a set of states that have a common invariant (mathematically codified as a predicate). The separation of contexts allows us to reason about each with a manageable scope. The mode-class represents a skeleton that links partial specifications via *cross-behavior* transitions.

In the following sections we first give an overview of the different uses of *modes* in software engineering, and then provide foundations for modal-behavior modeling that we use for behavior synthesis from scenarios.

3.1 Modes in Software Engineering

In the computing literature, there are two fundamentally different usages of the term 'mode': Modal Logic [11] and Hybrid Systems [12]. The former is used by logicians to describe so-called *necessity* and *possibility* used for multi-valued interpretations of mathematical logic. This use of the term modes is not related to the work presented here. The latter usage of 'mode' is to characterize a set of related behaviors. For example, a hybrid system exhibits a set of different behaviors, each of which can be characterized concisely by a set of continuous differential equations. These behaviors

are called *system modes*. The notion of *mode* that we use in this report is based on the ideas presented in [9] and has its origins in the theory of hybrid systems.

Ptolemy [12] is a computing framework for modeling computerized hybrid systems. Maraninchi and Remond [28] used modes to extend the synchronous language LUSTER with a *mode* construct which is, essentially, a discrete version of the *hybrid automata* [29].

Other examples in the Software Architecture community includes the use of modes in AADL [30], an Architecture Description Language where a component behavior is mapped to a set of modes. Hirsch et al. [31] used modes to identify different *structural configurations* of software components in a software architecture model, and this line of work has been improved in [32] to enable self-management in service-oriented architectures.

A few existing approaches attempted to formalize a mode-based specifications techniques. Modechart by Jahanian and Mok [33] is a specification language based on the RTL logic, however, it is not clear how they formulate the relation between a mode and a state. Paynter [34] described a viewpoint of relationship between states and modes, and identified four possible ways to adopt *modes* in describing system behavior. Although Paynter adopted the *non-exclusive* modes option to avoid the proving of invalid properties (see §2 in [34]), we believe that the idea of mode-classes [10] can avoid such problems and also promotes fundamental concepts such as separation of concerns. Moreover, having several mode-classes for the same system allows a state to belong to several modes (but each mode in a different mode-class) and achieving the same purpose of non-exclusion option adopted in [34].

3.2 Mode Abstraction

In this section we propose a formal model of the *mode* and *mode-class* concepts, underpinning the system prototype we synthesize in the sense that we use *mode* and *mode-classes* notions to structure scenarios specifications and synthesize an integrated model. A mode allows to specify a certain context within which one or more scenarios can be described and elaborated, and a mode-class completely partition the system space to a disjoint set of such contexts where scenarios can be specified exclusively within these contexts. Given a set of variables defining the state space of the system, a one or more mode-classes can be specified first to partition the state-space, and then scenarios are specified as described above.

3.2.1 Mode vs. State

An abstraction is one thing that represents several real things equally well [23]. We try here to establish a foundation of an abstraction of system behavior that can represent several possible *state-based* implementations. So, we begin by recalling the familiar concept of *system state*, or simply *state*, widely used in general Systems Engineering [26] and Model-Checking [35]. Simply speaking, a ‘state’ is a unique valuation of *all* system variables. With this definition, it is common also that a state is referred to as a *detailed state* or *concrete state* when compared to a more abstract representation. We use this definition of state here to establish a concrete level relative to which we then define the mode and mode-class abstractions.

DEFINITION 1 (Concrete State). Let $V=\{v_1, v_2, \ldots, v_n\}$ be the set of variables defining the system context. Assume that the variables $v_i \in V$ range over a finite set D, the domain of interpretation. The function $t: V \rightarrow D$ defines a set of possible *concrete states* $t \in T$ such that $t=\langle v_1 \leftarrow d_1, v_2 \leftarrow d_2, \ldots, v_n \leftarrow d_n \rangle$ and the variables valuations $v_i \leftarrow d_i$ is an *atomic proposition*[3].

■

The selection of the variables set V is application-specific. We assume the level of concreteness is characterized by the atomic valuation of *all* variables v_i in the form of $v_i=d_i$. For example, the state $t_i=\langle pr \leftarrow norm, ss \leftarrow inactive, sb \leftarrow off \rangle$ is a concrete state in the ESFAS system because every variable is assigned a value in atomic form. Note that a variable v_i may have identical value in several states, but the valuation of *all* variables, *together*, is unique across states.

DEFINITION 2 (Mode). Let $V=\{v_1, v_2, \ldots, v_n\}$ be the set of variables defining the context of a system with S possible states. Let Q be a predicate over V and interpreted in D. The subset of states $M \subset T$ is called a mode M that is characterized by the predicate Q such that

$$\exists\, t \in T,\ Q(t) \Rightarrow t \in M$$

■

That is, a mode M is a subset of possible system states that are satisfying some predicate Q. The predicate Q is said to be *characterizing the states in M*. This definition is related to the general notion of *predicate abstraction* [36] and we use it in the same sense as in [9].

As an example from the ESFAS system, the predicate $Q \Rightarrow ((pr>low \wedge pr \leq norm) \wedge (sb=off))$ characterizes the mode M that contains all sates where pressure is normal and safety activation is not blocked, regardless of the valuations of other variables in these states. The state $t=\langle pr \leftarrow perm, ss \leftarrow inactive, sb \leftarrow off \rangle$ is a concrete state satisfying Q and belongs to M.

There are two obvious means of abstracting detailed state information. One way is to omit a variable v from the predicate Q that characterizes M. In such case, Q asserts no information about v, and hence there could be as several *possible* states in M that have different possible valuations of v. Another mean of abstraction is to assign a range of values to a variable v appearing in the predicate Q. In this case, M will have as several possible states corresponding to different possible (atomic) valuations. It is needless to say that both cases can be applied to more than one variable. Using either of these cases is a designer choice.

When a predicate Q is satisfied by only one state, the mode characterized by Q is said to be a *singleton mode*.

REMARK 1 (Singleton Mode). A mode M characterized by a predicate Q is said to be Singleton Mode if

$$\exists! t \in T \mid Q(t).$$

[3] An *atomic proposition* is a formula with no deeper propositional structure.

That is, $M = \{t\}$.

■

This precisely differentiates between a state and a mode in our formulation of modes. More intuitively, a state must have *complete* context information, whereas that is not necessarily for a mode.

3.2.2 Mode and Mode-Classes

In order to structure a scenario, the designer will need to specify several contexts (or modes) that the scenario is supposed to exhibit such that each mode scopes a snippet of that scenario. These modes must be *disjoint* and (together) are *covering* the system state-space. Such a collection of modes is referred to as a *mode-class*.

DEFINITION 3 (Mode-Class). Let $V = \{v_1, v_2, \ldots, v_n\}$ be the set of variables defining the context of a system with S possible states. A collection of modes $MC = \langle M_1, M_2, \ldots, M_m \rangle$, characterized by a corresponding list of predicates $Q_{MC} = \langle Q_1, Q_2, \ldots, Q_m \rangle$, is called a Mode-Class *iff* each *state* $t_i \in T$ is in exactly one mode $M_i \in MC$. That is,

$$\forall\, t \in T, \bigoplus_{j=1\ldots m} Q_j(t)$$

■

A mode-class is a set of disjoint modes—scoping disjoint subsets of states—specified such that they cover the system state-space. In terms of scenarios structuring, on the one hand, disjointness of modes is useful because it minimizes redundancies in the scenario's steps and facilitates maintenance of the scenario artifact. On the other hand, the space coverage helps with spotting other contexts not covered by the scenario—allowing us to reason about what the scenario could do in these contexts and, accordingly, designers can adjust the scenario and/or the modes.

Let us pause at this point and discuss an important synergy between mode-classes and scenarios (or partial specifications in general). Since the *same system* space is possibly partitionable in several different ways (using different collections of predicates), designers may specify as several mode-classes as they see fit. Each mode-class partitions the state-space into a different set of modes. On the other hand, scenarios are generally perceived as several (possibly disjoint or overlapping) descriptions of the same system. This suggests that mode-classes allow for defining a variety of contexts where adding a new mode-class exposes emerging contexts. This provides a fertile environment to uncover possible gaps within which scenarios can be specified. At the same time, mode-classes organize those contexts in *classes* that facilitate independent elaboration and maintenance of partial specifications. To this end, designers can initially specify a set of mode machines and then use them as guidance in identifying independent contexts and write scenarios that fit under the scope of modes in a mode-class.. Moreover, as we will see shortly, mode-classes provide a disciplined way for merging a given set of scenarios, which is a major challenge arising in automated behavior synthesis from partial specifications [20].

Having defined the basic concepts of mode and mode-classes in previous sections, in the following discussions we use these concepts to define an (automata-based) abstract behavioral model that we will use as intermediate representation models in our synthesis process.

3.2.3 Mode Machines

At the mode's level of abstraction, the system behavior description is a standard transitions system between the set of modes in a mode-class. We refer to this model as an *Abstract Transition System* or, more intuitively, a *Mode-Machine*[4].

DEFINITION 4 (Mode Machine *MM*). For a space $T=\langle V, D\rangle$, a structure $MM = \langle V, Q_{MC}, MC, M_0, \varepsilon, \delta\rangle$ is called *Mode Machine*, where:

- V is a finite set of system variables.
- MC is a mode-class partitioning the space T.
- Q_{MC} is a collection of predicates characterizing modes in the mode-class *MC*.
- M_0 is the initial mode which includes the initial system state.
- $\delta \subseteq MC \times \varepsilon \times MC$ is the transition relation. A transition ℓ from M_i^a to M_j^a, denoted $M_i^a \xrightarrow{\ell} M_j^a$ is itself a predicate relation $\ell : V' \rightarrow V$ where the primed variables denote the variables after the transition.

■

The semantics of an *MM* is that of a standard transition system, with its nodes denoting modes instead of states. The initial mode M_0 is *the mode that includes the initial state of the system*. This can be understood directly from the definition of mode-class (Definition 3): since every state belongs to exactly one mode in the mode-class, then the initial-state of the system belongs to only one mode—the initial mode M_0.

Note that the exact sets of *possible* system states, each belong to a mode in an *MC*, are yet to be decided. The only information available about those states is the assertion formulated by predicates in Q_{MC}. This also applies to the initial-state. Despite the fact that the initial-state has not yet been decided, the sub-space (scoped by M_0) where this state belongs to is specified.

Designers should also provide the transition relation δ between different modes (in the same mode-class). Defining such a transition relation is a doable task at early stage of development, particularly because it is guided by the scenario to be described. Mode transition relation reflects the *inter-context* transitions, abstracting away from *intra-context* transitions that will be extracted from the scenario part which is scoped by that context.

3.2.4 Refinement of Mode-Machines

The ultimate target for mode-machine refinement is to reach a state transition system. One challenge is to successively partition the machine's modes in systematic manner, such that each mode is divided into *lower-level sub-modes* (or *sub-spaces*) of the state-

[4] We opted for this naming convention to avoid confusion with Modal Transition System, MTS [37], which uses the term 'modality' in the sense of Modal Logic.

space part scoped by that mode. Another challenge is to elaborate the machine's transition-relation such as to connect submodes belonging to different higher-level modes.

We address those two refinement challenges in the light of scenario structuring. In the following, the to-be-refined modes will be referred to as 'higher-level modes' or just 'modes', whereas modes resulting from the refinement will be referred to as 'sub-modes'.

3.2.4.1 Refining the state space

Refining the system's state-space requires each mode in the machine to undergo successive iterations of partitioning until we have all modes as singletons, while preserving disjointness and space-coverage constraints. Recall from REMARK 1 that A mode M is singleton if $\exists! t \in T \mid Q(t)$ where Q characterizes M. Each mode is partitioned successively into a set of *sub-modes*, constituting lower-level mode-class local to that mode, in the same sense as we have partitioned the whole system state-space into the very first mode-class.

> LEMMA 1 (Mode Refinement). Consider a Mode-Machine $MM = \langle V, Q_{MC}, MC, M_0, \varepsilon, \delta \rangle$. For every non-singleton mode $M_k \in MC$, characterized by predicate $Q_k \in Q_{MC}$, there exists a mode-class MC_k that partitions the space part scoped by M_k. MC_k is referred to as *refining* M_k, denoted as $M_k \prec MC_k$.
>
> ■
>
> PROOF. From Definition 3, the space scoped by M_k is partitioned by MC_k similar to the way the system space is partitioned by MC.
>
> ■

This means that a partitionable (i.e., non-singleton) mode $M_k \in MC$ can be refined to a local mode-class which is a set of disjoint sub-modes covering only the M_k's space part. The union of this set of sub-modes plus the M_k's peer modes in MC (excluding M_k itself) is also a system mode-class that refines MC. More formally,

$$\{(M^k_{i=1 \to n} \in MC_k) \bigcup (MC \setminus M_k)\} \quad \prec \quad MC$$

where M^k_i is the i-th sub-mode[5] in the n-modes MC_k. This is generalized in the following theorem.

---

[5] As a notation convention, we use superscript of an element to refer to the higher level item (set or collection) containing this element. We also use a subscript to an element to denote the position of this element in its containing item. For example, M^k_i denotes the i-th mode element in the MC_k, and MC_k mode-class corresponds to the is k-th mode M^l_k contained in some mode-class the MC_l, and so on , until we reach the highest-level (root) set MC that has no parent (and no sub/superscript).

THEOREM 1 (Mode-Class Refinement). Consider a Mode-Machine $MM = \langle V, Q_{MC}, MC, M_0, \mathcal{E}, \delta \rangle$ with $|MC|=m$, $m>1$. Let $M_j \in MC$, $1 \leq j \leq m$, be the non-singleton modes in MC. The mode-class MC^R defined by the union

$$MC^R = \bigcup_{(\exists\, j \leq m)} \{M^j_{i=1 \rightarrow n_j} \in MC_j\} \bigcup \{MC \setminus \{M_{\forall s \neq j}\}\}$$

is a refinement of MC, $MC \prec MC^R$, where:

- n the number of non-singleton modes in MC,
- $MC_j : M_j \prec MC_j$ are the (local) mode-classes refining the non-singleton modes M_j in MC,
- $n_j = |MC_j|$ the number of modes in a mode-class and $MC \prec MC^R$,
- $M_s \in MC$, *for all* $s \neq j$, *are the rest of modes in MC that are not partitioned in the refinement* $MC \prec MC^R$ *(for example, singleton modes in MC).*

■

PROOF**.** Direct from Lemma 1 and Theorem 1.

■

Theorem 2 can be explained in the reverse way: a mode-class MC is said to be refined by another mode-class MC^R (denoted as $MC \prec MC^R$) if

(1) there is one or more sets of modes $MC_j \in MC^R$, where $j \leq |MC|$, such that each set partitions a corresponding mode $M_j \in MC$, and

(2) $\{MC \setminus M_j\} = \{MC^R \setminus \{MC_j\}\}$, which means that the set of modes in MC that are not refined by the $MC \prec MC^R$ relation are appearing identically in MC^R.

The relation $MC \prec MC^R$ is intended to be stepwise refinement relation where not all modes in MC are necessarily refined, but at least one is. It is important to note that the there may be arbitrarily large possible refinements of the same mode-class. This directly follows from the fact that there may be arbitrarily large numbers of mode-classes partitioning the same state-space.

A successive refinement of a mode-class shall ultimately lead to a refined mode-class where all modes are singletons, each of which contains a unique state from S (the set of possible states). From refinement perspective, there can be several possible refinements, each of which results in a different set $T \subseteq \wp : V \times D$, and each refinement reflects a design alternative.

3.2.4.2 Elaborating the transition-relation

While the ultimate objective from refining the state-space is to reach a possible set of system states, the aim of transition relation is to reach a level of detail where each *possible* transition connects two states.

We may pause here to clarify what we mean by 'possible'. A state-transition is an action which causes the system to move from its current state to a new state. Some of these transitions may not take place because of physical constraints in the runtime environment. Another category of those transitions, while not constrained by the

runtime environment, but the system design may constrain them from taking place. So, we end up with a set of transitions that allowed by the runtime environment and are not constrained by the design (or requirements). Those transitions are the ones we term here as *possible transitions*. The same applies for the case of *possible states*. This argument is similar to the NAT and REQ mathematical relations in [38].

Mode-information is useful information at (1) design-time to help in a step-wise understanding and exploration of system behavior while keeping the whole system view and avoiding over-specification decisions. Also, analysis techniques applied to transitions systems such as model-checking and animations can be well applied to mode-machines because they have the same interpretation of standard automata. On the other hand, (2) it is useful at runtime too, for example, to help in identifying mode-specific tasks that are common to all scoped states. In this report, however, we focus on the application of modes at design time where we use them in structuring the contextual information of scenarios, and we leave the other potential uses of modes as future work.

Now we begin to formulate the step of mode transitions elaboration. This formulation will be used in our synthesis process at two milestones: (1) when constructing a number of mode-machines each of which corresponds to a mode-class along with its scoped scenarios, and (2) when merging these machines together in one integrated behavior model. In the sequel, the local mode-class that refines a mode M_q will be denoted as MC^q, and the i-th mode in MC^q will be denoted as M_i^q.

> **DEFINITION** 5 (Mode-Transition Elaboration) Consider $MM = \langle V, Q_{MC}, MC, M_0, \varepsilon, \delta \rangle$ and two modes $M_q, M_r \in MC$ are refined such that $M_q \prec MC^q$ and $M_r \prec MC^r$. *A transition* $M_q \xrightarrow{\ell} M_r$ is elaborated to a set of transitions $(\forall i,j:\ M_i^q \xrightarrow{\ell} M_j^r) \subseteq (MC^q \times MC^r)$ where $M_i^q \in MC^q$ and $M_j^r \in MC^r$.

■

3.3 Summary

In this section we provided formal abstractions for the concepts of *mode* and *mode-classes* [9, 10], and we defined an abstract form of a transition-system which we call a *mode-machine*. A mode-machine describes system behavior at some level of abstraction, and formally specified as a set of predicates, each predicate characterizing a mode, and a mode abstracts a disjoint subset of all possible system states. We also formulated a step-wise approach for refining a mode-machine to a more detailed mode-machine and eventually to a concrete state-machine model where a state is represented as complete valuation of system variables. We separated the refinement relation into (1) space-refinement where modes (or state-space parts) are successively partitioned into sub-modes until we reach singleton modes—each includes one state only—and (2) transitions-refinement where a transition between two modes is replaced by a set of finer transitions between pairs of sub-modes (and eventually pairs of states) resulting from space-refinement of these two higher-level modes.

This automata-based formulation establishes a basis for structuring scenario-based requirements specification by augmenting (or scoping) scenarios with contextual

specifications in the form of mode-classes. In the next section we detail a behavior synthesis process, accepting *structured-scenarios* as input and producing as output a state-based system prototype amenable for further analysis and verification, using stepwise refinement of mode-based models as intermediate representations throughout the process flow.

4. SYNTHESZING MODAL BEHAVIOR

This section details a synthesis process to generate an integrated behavioral model, given a set of scenarios and mode-classes as input. The steps of the synthesis process follow the formal semantics of modes, mode-machines, and their refinement to final state-machine model (see Section 3). This ensures accuracy of generated prototype, thus providing validation of the process. Figure 4 shows a schematic diagram showing major phases of the process.

4.1 Process input

The synthesis process we propose is intended to be part of the software requirements specifications and analysis cycle. The process semi-automates the construction of automata-based system prototype, amenable to further analyses such as model checking [35].

Input to the process consists of three related types of specifications:

(1) A group of scenarios. The scenario format is assumed to be a Sequence Diagram *SD*—the simplest form of a scenario. Actions in an *SD* are annotated with (possibly incomplete) state information commonly known as *pre/post conditions* (cf. OCL [39]). We assume pre/post conditions annotations are in form of a vector $<v_1, v_2, ..., v_n>$ which we refer to as the *state vector*, where v_i are system variables. The special symbol '?' is used in as the value of a variable v_i to indicate that the value of this variable is unknown.

An advantage of our approach is that *we do not constrain the designer to specify complete state information*. Sometimes a variable may be absent from pre/post state vectors, and in another case a variable may have a range of variables instead of atomic valuation. In either case, we assume designers have not decided yet about exact valuations of some variables. This is normal and typical situation at early stages of development.

As we sill see shortly, incomplete state information protects the requirements from over-specifications resulting from early design decisions, and also facilitates the merging of machines independently generated from different *SD*s.

(2) The second type of specifications is a group of mode-classes, *MC*s (see Section 3), that augment the *SD*s such that each mode-class scopes a subset of these *SD*s. Designers are expected to specify *MC*s for the purpose of providing contextual information for each *SD*. The relation between *MC*s and *SD*s inputs is clarified in the following two points:

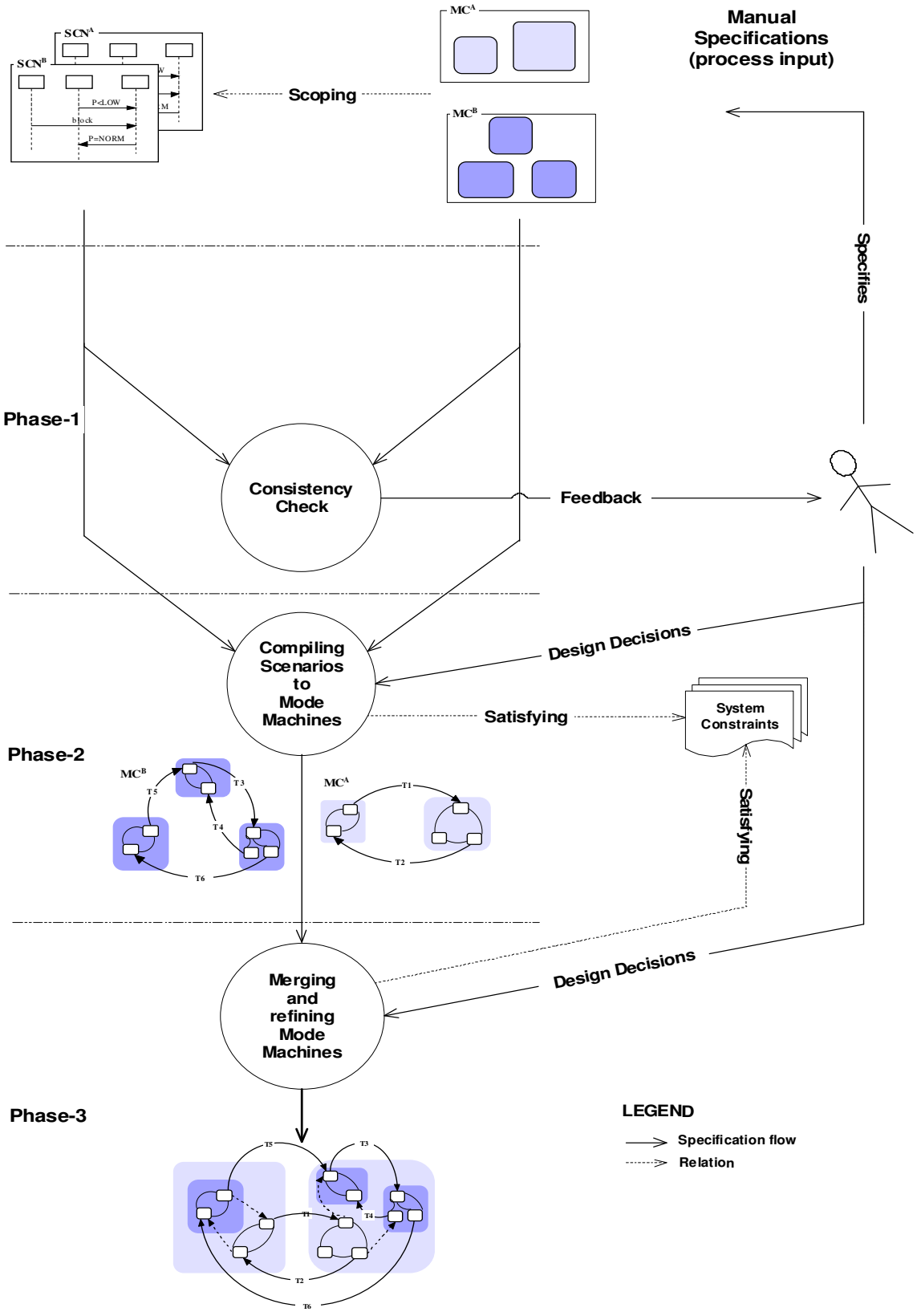

Figure 3: Mode-based behavior synthesis process

(a) Although it is correct to specify *MC*s such that every *SD* is scoped with several modes—one mode from each mode-class—we restrict input

specifications such that an *SD* is scoped (as a whole) by one and only one mode from a *MC*. This means that every set of *SD*s are associated with an *MC* such that each *SD* is scoped by a mode in that *MC*. Of course an *SD* will be crossing the boundaries of modes in another *MC*. It is worth noting that a mode may not be associated with an *SD*, however every *SD* is associated with exactly one mode.

(b) On the other hand, we do not see (so far) any motivation for defining several *SD*s in the same mode. If necessary, when a mode happens to be scoping several *SD*s, this mode may be broken down into as several sub-modes as the number of *SD*s.

In summary (syntactically) we assume a one to one relation between *SD*s and modes, and (semantically) the set of *SD*s scoped by a mode-class are, together, a *structured form* of a longer scenario.

(3) The last of type of input is a set of system constraints that assert undesirable system behaviors—those behaviors that must not be exhibited by the system under prototyping. Our process exploits these constraints to help in decisions about potential transitions between modes.

The rest of this section details the individual phases of the synthesis process along with associated algorithms accompanied by their complexity analysis.

4.2 Phase 1: Preparing Input Specifications

This phase performs consistency checks on two aspects:

- Checking legitimacy conditions of a mode-class (see Definition 3).
- Checking consistency of each scenario with its mode.
- Connecting the modes in the same mode-class with general transitions.

(1) Mode-class legitimacy. Conditions for mode-class legitimacy were already expressed (on an arbitrary example) by Parnas as "Domain Coverage Theorem" and "Disjoint Domain Theorem" [40]. A demonstration of automatic check of these theorems has been done using PVS [41]. In our ongoing tool support work, we are integrating PVS in similar way to [42], where a tool for an automatic check of Tabular notations is integrated with PVS.

(2) Scenario and mode consistency is checked by examining the mode's predicate against the state-vector valuations (i.e., the pre/post conditions) at every action in the scenario's *SD*. The *SD* is consistent with its mode if no variable valuation at any pre/post condition is contradicting with the mode's predicate. Listing 2 shows a procedure for this step.

Algorithm SD_Mode_Consistency(V, SD)

Input:

(a) V is a list of n system variables v.

(b) Scenario SD=<ACT,Q_{SD}> where:
$Act_j \in ACT$, $j \leq |ACT| = m$, is the set of actions in SD.
$Act_j = <Pre_j, L_j, Post_j>$ is an action in SD where: Pre_j and $Post_j$ are predicates formed as conjunction of

valuations of one or more variables v. L_j is string label of Act_j.
Q_{SD} predicate of the mode scoping SD.

Note: as notational convention, for any predicate Q we use Q(v) to denote the valuation of v as specified in Q. if a variable is not specified in Q, then Q(v)=undef.

Procedure:

```
1 FOR j=1 to m
2  FOR i=1 to n
3   IF (Q_SD(v_i) = {∅} OR Pre_j(v_i) = undef) THEN NEXT j;
4    IF Pre_j(v_i)∈ P_SCN(v_i) ELSE RETURN 0
5 RETURN 1
```

Listing 1: Checking scenario consistency with its mode

The basic idea of that procedure in Listing 1 is as follow: For each *SD* involving a set of *m* actions *ACT* and scoped by a predicate Q_{SD}, if a variable v_i is valuated in Q_{SD} with some range, then any valuation of v_i in any of the *SD*'s pre/post conditions must be within that range of v_i specified in Q_{SD}.

In this sense, Q_{SD} plays the role of *invariant* for all of the pre/post conditions in the *SD*. Ideally, but not necessarily, variables involved in a mode's predicate will be independent from the pre/post conditions. In our experience, it is better to specify a mode-class (the *SD*'s scoping predicate) with respect to variables which are different from other variables used in specifications of sub-modes (*SD*'s pre/post conditions). This allows us to model system behavior by varying the sub-modes' variables, while holding other (scoping) variables as invariants. The key point is to decide which variables to use in either case, and this is an application-specific decision.

(3) Connecting higher-level modes with abstract transitions. In this step, we connect the higher-level modes in the same mode-class with *general* transitions such that: for two modes M_1 and M_2, characterized by predicates Q_1 and Q_2, respectively, a transition *from* M_1 *to* M_2 is added and labeled by Q_2. Similarly, another transition from M_2 to M_1 is added and labeled with Q_1. Due to disjointness of modes, the predicate of destination mode represents the *variables' valuations change* that triggers any transition to it.

Complexity Analysis. For step (2), the algorithm in Listing 1 runs in a polynomial order O(|*V*|*|*ACT*|), a where |*ACT*| is the number of actions in the *SD* and |*V*| is the number of system variables. While a scenario may have a relatively long list of actions, the number of variables is typically much smaller and so the complexity reduces to a O(|*ACT*| log |*ACT*|). Step (3) is expected to run in constant time, compared to step (2), because it iterates over modes in every mode-class, with each mode correspond to an *SD*, and the total number *SD*s is negligible compared to the total number of actions in those *SD*s. The time effort of the first step is dependent on the response time of the theorem prover which also expected to be small as the number of system variables are kept manageable.

4.3 Phase 2: Constructing mode machines

The input to this phase is the same as input to Phase 1, with specifications' inconsistencies are resolved. In this phase, we translate an *MC*, combined with the *SD*s it scopes, to a mode-machine. This is applied for every *MC* in input specifications, resulting in a set of independent mode-machines to be merged in Phase 3.

A machine constructed in this phase has two-level hierarchy of modes and sub-modes. A higher-level mode-machine consists of those modes belonging to the input *MC*—each mode scopes an *SD*. In each of these modes, there is a set of sub-modes—constituting a *lower-level mode-machine*—corresponding to pre/post conditions' predicates specified in the *SD* scoped by that mode. For brevity, we will refer to this lower-level machine as the *submode-machine*, and we will refer to the higher-level machine as just a mode-machine.

For every mode-class *MC*, and the *SD*s it scopes, we construct one mode-machine with each of its modes containing one submode-machine. To do this, Phase 2 performs the following steps:

(1) For every mode in *MC*, we compile the corresponding *SD* to a submode-machine.

(2) Modes in *MC* are then connected together to construct the higher-level mode-machine as follows: for every two modes M_1, $M_2 \in MC$, characterized by predicates Q_1 and Q_2 respectively, we add a transition $T_{(M_1 \to M_2)}$ labeled by the predicate $\neg Q_1 \wedge Q_2$. A symmetric transition from $T_{(M_2 \to M_1)}$ labeled with $\neg Q_2 \wedge Q_1$ is also added. Predicates $\neg Q_1 \wedge Q_2$ and $\neg Q_2 \wedge Q_1$ are corresponding to *syntactic constraints* (see Theorem 4) on transitions between modes M_1, M_2. The initial mode is assumed to be indicated by designer in the definition of mode-class, which is manageable because typically the number of modes in a mode-class is small. This is compared to other related approaches where designers have to specify an initial *state*, which is more difficult to do at early stage of development. Our approach relieves designers from this early decision by instead specifying a *state-space part* (i.e. mode) where the initial state is expected to lie.

(3) The final step in Phase 2 is to find possible transitions between sub-modes belonging to different modes in *MC*. The possible transitions must satisfy the *syntactic-constraints* transitions made in the previous step, or otherwise they are not added to the machine.

(1) Compiling scenarios to (sub) mode-machines. Translating an *SD* to a mode-based model (instead of *state*-based) relieves designers from making early decisions about *complete state-information* at the *SD*'s pre/post conditions—allowing an incremental exploration of (and building knowledge about) system behavior. On the other hand, it provides an opportunity for possible interleaving between final states in the final states-machine model (see Phase 3). As an example, the ESFAS mode-classes MC^{pr} and MC^{sb} are shown visually in Figure 4.

In this step, every *SD* is translated to a submode-machine

Listing 2 shows a procedure to perform this step.

```
Algorithm CompileSD(V, SD, MM)
Input:
```

(a) V is a list of n system variables v_i.

(b) SD=<ACT,Q> is a sequence diagram where:
ACT is the list of actions in SD (ordered by precedence in SD's flow).
$\forall Act_j \in ACT$, $j \leq |ACT|=m$, Act_j=<Pre,L,Post>:
Pre and Post are two predicates corresponding to pre and post conditions, respectively, that are associated with Act_j. The Pre and Post predicates are formed as. L_j is string label of Act_j.
Q predicate of the mode scoping SD.

(c) MM=<Q,MC,T> is a mode-machine data structure to store the compiled machine, where:
Q is a predicate scoping MM, MC is a list of modes constituting MM's mode-class, and T is a list of transitions between modes in MC. For all $T_r \in T$, T_r=<M_s,Q,M_d> connects M_s (src) and M_d (dest.) modes, such that <M_s,M_d>$\in$MC×MC, and Q is the event-predicate triggering T_r.

NOTES:

Any predicate Q is represented as a conjunction of valuations of one or more variables $v_i \in V$, where Q(i) denotes the valuation of v_i as specified in Q. When v_i is not specified in Q then $Q(v_i)$=undef.

Procedure:

```
1  k=1;
2  MM.Q = SD.Q;
3  FOR j=1 to m
4   MM.MC(k).Q   = SD.Act(j).Pre;
    MM.MC(k+1).Q = SD.Act(j).Post;
5   linkModes(MM.MC(k), MM.MC(k+1));
6   k++;
7  NEXT j;

8  WHILE k≥1
9   j=1;
10  WHILE j<k
11   FOR i=1 to n
12    IF MM.MC(j).Q(i) ≠ MM.MC(k).Q(i)
13    THEN NEXT i;
14    IF i=n THEN
          mergeIdnenticalModes(MM.MC(j), MM.MC(k));
15   NEXT i;
16  NEXT k;
17 NEXT j;
```

Listing 2: Translating a scenario to a mode-machine

The procedure in Listing 2 does the following majors tasks:

- Steps 1-7 store pre/post conditions as modes into the M_{SD} mode-machine and add a transition—corresponding to the difference between pre and post conditions—between these modes using the function **`linkModes()`**. Essentially, this builds the initial mode-machine which is a sub-machine to the mode scoping *SD*.
- Steps 7-17 searches this mode-machine for identical modes and merge them together. Two modes are identical if they are characterized by identical predicates. The function **`mergeIdenticalModes()`** merges two identical modes in one mode that has the union of the two modes' incoming and outgoing transitions.

As an example, Figure 4 shows the refined mode-machine corresponding to mode-class MC^{pr} in the ESFAS system. The lower-level machines correspond to *SD*s scoped by the mode-classes. For space reasons, we have omitted these *SD*s, but they still can be straightforward understood from the figure. Nodes in those machines are (sub) modes corresponding to pre/post conditions in the compiled *SD*s. The higher-level machines are corresponding to MC^{pr} and MC^{sb} classes.

Note that in MC^{A}, transitions between modes that appear in Figure 2 are refined into transitions between lower-level sub-modes. The next step in this phase will explain how those transitions are refined.

A final noteworthy point about the machine in Figure 5: according to Theorem 2 (Section 3) a sub-machine of a certain mode must be a *refinement* of this mode. However, it is not the case with the sub-machines in Figure 5 because an *SD*'s pre/post conditions are not necessarily expressed by the designer to completely covering the

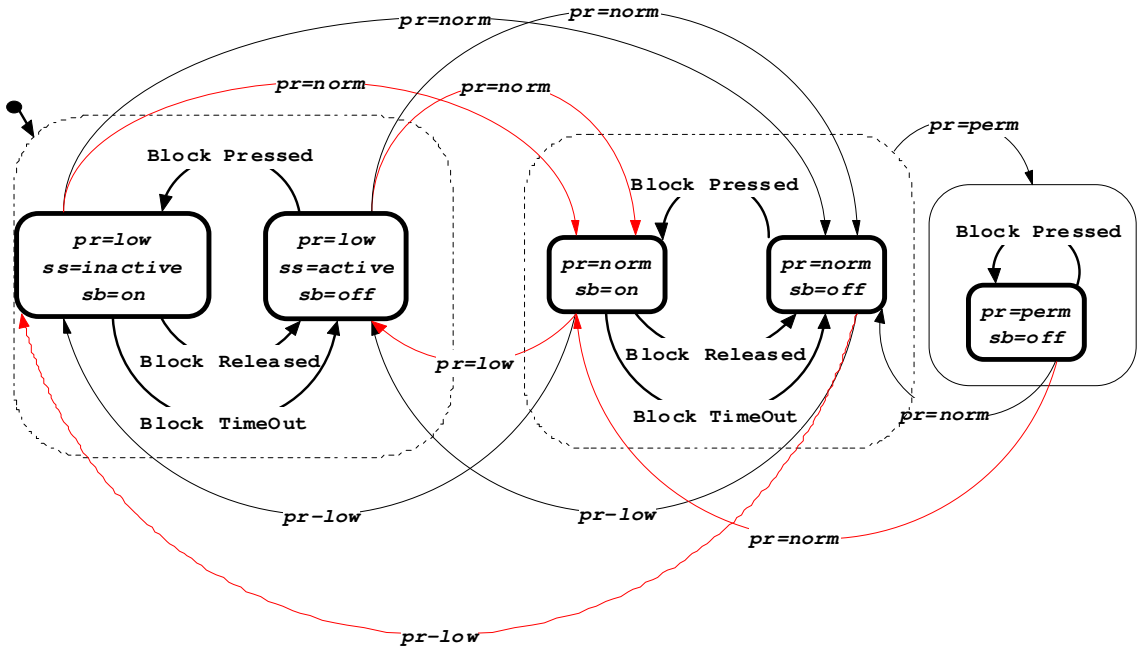

Figure 4: ESFAS mode-machine MC^{p} after Phase 2. (For brevity reasons, we omitted the variables with unknown value '?').

space part scoping this *SD*. This semantic problem will be resolved in Phase 3 where all (high-level) mode-machines are merged together such as they partition each other, resulting in one integrated mode which is a *common refinement* of any of the machines.

(2) Refining transitions. In this step, we refine transitions between the scoping modes––the higher-level modes that scope *SD*s. In terms of scenarios specifications, these transitions are connecting different scenarios that are scoped by different modes in the same mode-class. Abstraction as the key for invariant verification

For two modes M_1 and M_2, characterized by predicates Q_1 and Q_2, refining the transition $M_1 \xrightarrow{\ell} M_2$ is the process of finding all possible (but legal) transitions from sub-modes in M_1 and those sub-modes in M_2 For example, in Figure 2, the transitions between higher-level modes in MC^{A} machine are replaced by transitions between submachines of these modes. Figure 5 shows the refinement of transitions between modes M_1^{pr}, M_2^{pr} (refinements of other transitions are not showen for space reasons). The procedure in Listing 3 performs this refinement step.

Algorithm RefineTransitins(MM)

Input:

(a) MM=<Q_{MM},*MC*,T> is a mode-machine where: Q_{MM} is a predicate scoping MM, *MC* is a list of modes constituting the machine's mode-class, and T is a list of transitions between modes in *MC*. For all $T_r \in T$, T_r=<M_s,Q,M_d> connects M_s (src) and M_d (dest.) modes, such that <M_s,M_d>$\in$ *MC*×*MC*, and Q is the event-predicate triggering T_r.

(b) System constraints C

(c) T_{TEMP} is temporary list of transitions

Note:

(1) The predicate Q_{MM} scopes the whole system because M a system-level machine. So Q_{MM} is not meaningful and used here only for notational reasons.

Procedure:

```
1  WHILE s=1 to m
2   WHILE d=s to m
3    T_TEMP = detectPossibleTransitions(M_s, M_d);
4     FOR i=1 to sizeof(T_TEMP)
5      IF Satisfied (T_TEMP(i).Q, M_d.Q_j)
6       IF Satisfied (T_TEMP(i), C)
7        IF designerAccepts(T_TEMP(k))
8        THEN addTransition(T_TEMP(k), M_s, M_d);
9   Next d;
10 Next s;
```

Listing 3 Refining transitions between higher-level modes.

The function **`detectPossibleTransitions()`** checks for differences (in variables' valuations) between the predicates scoping M_s and M_d. If a variable appears in both predicates with different values, then there is a *potential transition* between M_s and M_d. Detected transitions are then put under three complementary tests:

(a) A transition is allowed only if it is valid (Theorem 4). That is, a transitions is allowed if it satisfies Q_d which is the predicate characterizing the destination (sub) mode. In general, a transition from (or to) a mode must be valid for all sub-modes of this mode. This is the case with syntactic transitions we added in Phase 1 step 3. However, designers may (manually) specify a transition that thought be valid, but it is not. For example, in the mode-class MC^B Figure 2, the transition "`Block timeout`" is manually-specified transition, and it must be checked against the source and destination modes' predicates. In short, this step is intended to check the validity (according to Theorem 4) of manually-specified transitions.

(b) A transition is allowed only if it satisfies the (user-defined) system properties or constraints `C` (part of the inputs to the synthesis process).

(c) If a transition passed the two checks above, then it must be finally checked by a designer to see if it is *realistic* or not. It is important to understand the system's actual behavior so as to avoid over- or under-specifications. As an example of this type of transitions, consider the red-colored transition *pr=norm* in Figure 5 between the states <*pr=low,ss=inactive,sb=off*> and <*pr=norm,ss=*?,*sb=on*>. This transition is syntactically correct and also it does not violate any of the ESFAS constraints given in Section 2.2. However, the designer can easily detect that this transition is not safe because it allows the ESFAS system to *automatically block the safety activation* (*sb=on*) which is very unsafe to leave such critical action to be automatically decided by ESFAS. The designer then would disapprove such a transition to be added to the system behavior model, and may also add it to the set of system constraints *C*.

Despite the fact that an event is syntactically correct (and does not violate system constraints) it may be an over-specifications because such an event stet leads verification tools to incorrectly indicate false errors that do not happen at runtime [35]. The same situation occurs with under-specifications, where an event could happen at runtime but has not been specified in the model.

It is important to note that the transitions to be derived manually are *user-friendly* because they are still at the level of modes, not states. Moreover, a proper tool support can help the designer to stop this transition-detection process at some given level of detail.

These constraints are expected to reduce the number of manually-resolved transitions to a manageable list to be resolved (semantically) by the designer.

Complexity Analysis. This phase takes the largest effort in the whole process. For step (1), the algorithm in Listing 2 has a complexity of $O(|ACT|^2)$ calculated as follows:

- steps 1-7 has a $|ACT|$ number of iterations. The function **`linkModes()`** takes constant time because it adds just one transition corresponding to the difference in variables' valuations in the two modes.

- steps 7-17 has a (k*j)/2 number of iteration, but since k=2*|*ACT*| (two modes per action) then the number of iterations are $|ACT|^2$. The function **`mergeIdenticalModes()`** takes constant time because each of the two modes has exactly two transitions (due to the sequential nature of a scenario).

For step (2), algorithm in Listing 3 makes a heavy use of theorem prover to check satisfiability of system constraints. Moreover, the algorithm involves designer's decisions. Assuming all external effort is constant, the algorithm executes in $O(|V|*|ACT|^2)$ where **`detectPossibleTransitions()`** involves |*V*| iterations.

So, Phase 2 runs in is a polynomial time of $O(|ACT| + |ACT|^2 + |V|*|ACT|^2)$ that reduces to $O(|V|*|ACT|^2)$ and further to $O(|ACT|^2 * log|ACT|)$ if |*V*| is kept much less than |*ACT*|.

4.4 Phase 3: Merging and refining mode-machines.

While the first two phases are concerned with refining each mode-machine independently, this phase merge those refined machine together in one integrated model. This is done in by (1) pair-wise merging of refined machines into one integrated mode-machine, and (2) then a state-information refinement is done on all modes in this integrated model so that each mode is refined to a *singleton* by adding the missing state-information. The second step is a yet more chance for designers to explore the state-space and add possible states as we will see shortly.

The two steps are summarized as follows:

(1) Merging a pair of mode-machines M^A and M^B involves two sub-steps: (a) determining the intersection between modes in every modes pair $< M_i^A, M_j^B > \in M^A \times M^B$, and (b) adjusting the related transitions in M^A and M^B to accommodate the resulting intersection-modes. This is repeated for every two machines until we reach to one integrated system model.

(2) Each mode in resultant system model is then refined by adding state-information so that each mode becomes a singleton mode—a mode containing one only one state (see Remark 1). A *mode* reduces to a *state* if the mode's predicate is equivalent to a complete variables' valuation.

(1) Merging a pair of mode-machines. The merge process is motivated by a fundamental principle of mode-classes: they are behavioral-classifications of the *same system* and specified from *different dimensions*. This means that a mode $M_i^A \in M^A$ must overlap with at least one mode $M_j^B \in M^B$, and it possibly overlaps with as several as all modes in M^B. It is straightforward to note that in case M_i^A overlaps with only one mode from M_j^B, then this implies that $M_i^A \subseteq M_j^B$. This remark helps to optimize the procedure performing this step, shown in Listing 5. The procedure has two components, explained as follows:

for every pair of machines M^A and M^B such that $M^A_{i=1\to n} \in M^A$ and $M^B_{j=1\to m} \in M^B$:

(a) First, identifying possible overlap between every modes pair $< M_i^A, M_j^B >$. If M_i^A overlaps with M_j^B then we modify M_i^A to be $M_i^A \setminus \{M_i^A \bigcap M_j^B\}$. Similarly we modify M_j^B to be $M_j^B \setminus \{M_j^B \bigcap M_i^A\}$, and finally we create a new mode $\{M_i^A \bigcap M_j^B\}$

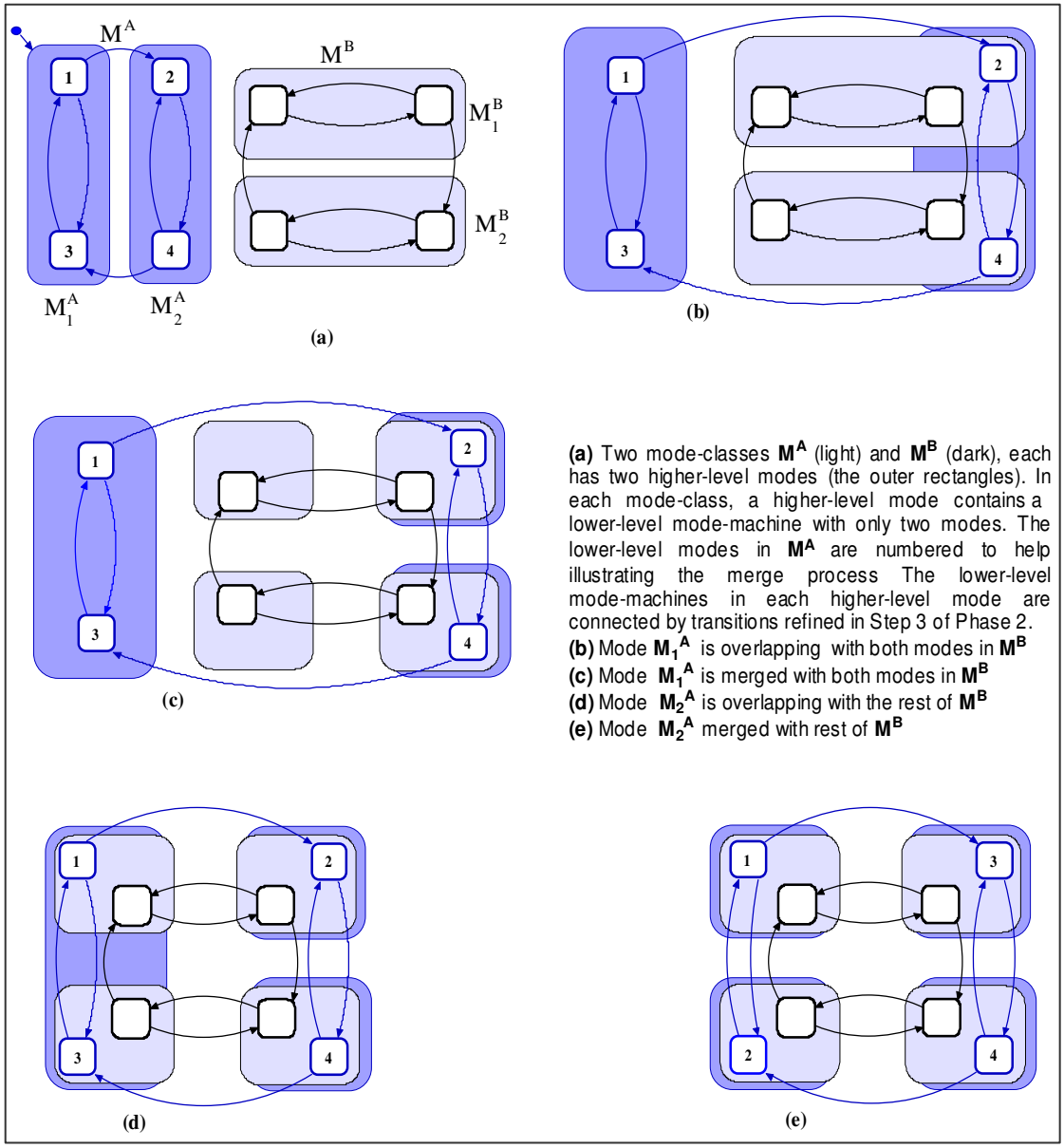

Figure 5: Illustration of mode-classes merging process

that scopes their intersection. Sub-modes of M_i^A and M_j^B that are lying in the overlap space are assigned to the newly created mode. Moreover, the new mode is updated by those existing transitions connected to the overlap sub-modes and are crossing the boundaries of the new mode.

(b) Second, detecting and resolving possible transitions between overlapping modes. In terms of scenarios, an overlap two between two modes M_i^A and M_j^B indicates that the two scenarios scoped by M_i^A and M_j^B have potential for transitions' connections between their sub-modes (or states in the final model). For example, one scenario may be starting after the other, or even the execution may be jumping between states in these scenarios. This avails a further opportunity for enriching system behavior.

Algorithm mergeMachines(M^A , M^B **)**

Input:

(a) Machines $M^A = <Q^A, MC^A, T^A>$ and $M^B = <Q^B, MC^B, T^B>$ where $M^A_{i=1 \to n} \in M^A$ and $M^B_{j=1 \to m} \in M^B$.

(b) System constraints C

(c) T_{TEMP} is temporary list of transitions

(d) M_{TEMP} :temporary mode data structure to hold overlap space and its submodes

Procedure:

```
1 WHILE i=1 to n
2   WHILE j=1 to m
3     M_TEMP = determineOverlapSpace( M_i^A , M_j^B );
4     IF  M_TEMP ≠ {∅}
5     THEN T_TEMP = detectPossibleTransitions( M_TEMP , M_i^A );
           connectMachines( M_TEMP , M_i^A );
           T_TEMP = detectPossibleTransitions( M_TEMP , M_i^B );
           connectMachines( M_TEMP , M_i^B );
6  Next j
7 Next I;
```

Listing 4: Merging mode-machines M_i^A and M_j^B

Figure 5 shows a general case example of merging two machines M^A and M^B, where each machine consists of two modes. (For brevity, potential transitions between newly created modes are not shown in Figure 5).

(2) Final modes-to-states refinement. In this step, the single integrated mode-based system model built so far is refined to a state-based model. This is done by iterating over all modes in this model and *completing the missing state-information* so that every mode reduces to one state—a singleton mode. The missing state-information in a mode is either a missing variable (does not appear in the mode's predicate) or a variable defined with a range of values. In the former case, the missing variable is presented to designer to make a decision regarding its value. In the latter case, the designer must decide either to consider the value-range as one atomic value, or alternatively to create several states in the mode so as to accommodate the variation of this variable within the specified range. In either case, considering the variable's valuation in neighboring modes (or states) is helpful. Once every mode is resolved to state(s), the machine model will have complete state information. It is important to note that mode-hierarchy information (i.e. modes' predicates and modes-sub-modes relations) are retained in the model for design-time analysis. At runtime, however, this information is (formally) redundant such that the system exists in one and only one state at a time—this distinguishes our model from hierarchical state-machines formalisms. However, it is possible for an implementation to make use of mode-information for optimization purposes (e.g. performing a task which is common to all states in a mode, instead of repeating it each time a state in this mode is entered).

Complexity Analysis. Algorithm in Listing 4 has a complexity effort of $O(3*|V|*|SDs|^2)$ where $|SDs|$ is average number of scenarios per mode-class input to

the process (see process inputs Section 4.1) and each of the functions `determineOverlapSpace()` and `detectPossibleTransitions()` involves |*V*| iterations. This complexity is acceptable because the number of scenarios in a mode-class is equal to the number of modes in this mode-class (see section 4.1).

4.5 Summary

We have described in this section a synthesis process that takes as input a collection of scenarios snippets organized under the scope of a corresponding collection of mode-classes. The intermediate models of the process are maintained as mode-machines until the last step. As we showed in the individual phases, this helped to reduce the effort and complexity that would be required by the process procedures if we moved directly to state-models from the outset. The process involves some interactions with the designer, particularly when connecting modes together and when creating new modes in the merge process.

5. DISCUSSION AND RELATED WORK

A wide range of techniques have been proposed to automate the construction of behavioral models out of partial specifications in general and scenario-based specifications in particular.

To the best of our knowledge, no prior approach addresses the structuring of contexts in partial specifications in terms of providing coverage of behavior space and prevents inter-contexts traversing. However, on the other hand, the extraction of contextual information from a given specification is often done [43],[44]. In the following discussion, we discuss these approaches, and the others related ones, with respect to each issue of behavior synthesis.

On partiality. A common direction for addressing partiality is to enrich the machines, independently-generated from scenarios, with *possible* behaviors and delay refinement decisions at the merging phase. Uchitel et al. [45] (and improved in [20]) use special logic to capture the possible behaviors. Krka et al. use the same technique but to generate component-level models. Another range of techniques (c.f. [23-25]) *infer* behaviors from the given specifications. A common denominator of these approaches is the use of *bare* scenarios without a structuring framework. This distracts the synthesis process from generating correct and structured models to issues related to partiality itself. We consider partiality as symptom of the main problem of *a lack of a framework within which the partial specifications are described*. We assume simple form of scenarios as input; however we augment it with mode specifications to provide structuring and scoping of each scenario.

Other proposals accept more expressive forms of scenario than an *SD* as input. Uchitel et al. [19] synthesize behavioral models from Message Sequence Charts [5] to detect implied scenarios. Whittle and Jayaraman [46] use the recent Interactions Overview Diagrams IOD in UML 2.0 [4] to generate Statechart-based designs. Though specified at a higher-level, these forms of scenarios specification express *control-flow* information between lower-level scenarios— a yet more operational view of the system with some control information similar to that of flow-charts. The *higher-level* feature in such specifications does not address coverage or integration between the lower-level

scenarios. Our approach is distinct from these proposals by structuring the specifications in the state-space rather than a flowcharting structure.

On abstraction. This direction of work augments the partial specifications with auxiliary specifications in some form of abstraction to help overcome complexities in the synthesis process. Sun and Dong [43] combine Live Sequence Charts, LSC, with Z specifications. They use predicate abstraction techniques to find predicates in LSCs. Although they extract abstract state-information similar to modes in our approach, the extracted information does not guarantee scoping (coverage) of state-space as we do by mode-classes. Also the mode-classes help designers to structure the specifications, and provide feedback about the underspecified aspects of scenarios, hence helping for earlier elicitation of more requirements. Another recent work of de Caso et al. [44] automatically constructs useful FSM-like abstractions from specifications based on pre/post conditions. Although the extracted abstraction is promising for its intended validation purposes, it is similar to [43] in the sense that it is extracted from a given specification and does not guarantee space-coverage as we do with mode-based specification.

We use modes as *user-defined* specifications rather than extractions from scenarios. We have provided a formal basis for specifying modes and partitioning the system state-space from different dimensions, based on the idea in [10], and sticking to standard mathematical concepts without using any unfamiliar formal philosophies– thus allowing the model to be applied to a wider range of applications. In this sense modes can be applied to other partial specifications such as goals and properties. In the general sense, we envision mode-based design to be used as a design methodology similar to Statecharts, but avoiding the mix between *higher-level state* and *mode*.

On Synthesis Process and tooling. Our proposed process is quite interactive with the designer. The level of designer involvement depends to a great extent on the availability of system constraints (such as safety properties) to cut down the possible transitions. Agreeing with others (cf. [7]), we consider this to be a positive thing as it contributes to designer's experience with the system. The feedback to the designer helps to flesh-out the system details as early as possible, and yet presents the information in abstract form (modes) without overwhelming details.

Several of the related approaches assume designer's involvement at some stage in the synthesis process. Krka et al. [8] assume nondeterministic transitions appearing in Phase 2 of their synthesis process. In Mäkinen and Systä [47], their MAS tool generates component-level behavior. Based on grammatical inference, MAS asks the designer *trace questions* in order to avoid undesirable generalizations. (A trace question is a path in the state machine local to a specific agent). MAS focuses on single agents; generalization must therefore be done independently for each software agent. Trace questions may be quite hard for designers to understand as they do not show global system behaviors. The same applies in Damas et al. [48], where the designer is presented with *scenario questions* to classify it as positive or negative. A related assumption about designer-involvement is also made in other inductive-learning approaches such as [49].

when involving the designer, the key point in our opinion is not to present the designer with too detailed information to comprehend. In our approach, we present *mode-level* information to the designer, and the designer still has the option to delve into more details when needed.

On the other hand, generating and manipulating mode-machines throughout the synthesis process (instead of state-machine as in almost all related proposals) avoids a possible explosion of states. The disjointness of modes in the same mode-class helps to avoid non-deterministic transitions between lower-level modes when merging mode-machines. Finally, partitioning the state-space allow designers to relax the assumption that a scenario must start at an initial state so that they can describe scenarios that start at any state within its scoping mode.

6. CONCLUSIONS

We propose a framework to support rapid prototyping of system behavioral model from partial specifications, in the form of sequence diagrams, and mode-based specifications. We use the classic idea of *modes* and mode-classes to structure the partial specifications so as to provide adequate coverage of system requirements at early stages of development. Modes abstraction is a powerful concept for scoping and decomposing system behavior. The flexibility provided by mode-classes for partitioning the behavior from different viewpoints (or *dimensions*) allows designers to, intuitively, define contexts and specify detailed behaviors within those contexts, in a stepwise refinement process. These behaviors may be, for example, precise functions [27], goals [1] or scenarios, as the case in this report.

We complemented this framework with a (semi) automated synthesis process to generate behavior models of sequence charts that are structured and scoped by mode-classes. The resulting models are standard automata-based and are amenable to further reasoning using off-the-shelf model-checkers or simulation/testing tools. The proposed framework supports designers by giving feedback about unforeseen aspects—situations or contexts not covered by the existing scenarios—that can be used to elaborate those requirements and discover new requirements early in the development process.

Future developments of our approach include (1) implementing a tool for the synthesis process. (2) Integrating an off-the-shelf theorem prover to support complex predicate specifications so as to relax the assumption we made in Section 4.1, (3) Evaluate the approach on a real world system to evaluate its effectiveness as a design methodology and to evaluate the time complexity of prospective tool.

ACKNOWLEDGEMENT

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero – the Irish Software Engineering Research Centre (www.lero.ie).

REFERENCES

[1] Letier, E., Kramer, J., Magee, J. and Uchitel, S. Deriving event-based transition systems from goal-oriented requirements models. *Automated Software Engineering*, 15(2)2008).

[2] D. Amyot, S. Ghanavati, J. Horkoff, G. Mussbacher, L. Peyton and E. Yu Evaluating Goal Models within the Goal-oriented Requirement Language. *International Journal of Intelligent Systems* (2010).

[3] Kazhamiakin, R., Pistore, M. and Roveri, M. Formal Verification of Requirements using SPIN: A Case Study on Web Services. In *Proceedings of the Software Engineering and Formal Methods* (2004), [insert City of Publication],[insert 2004 of Publication].

[4] OMG UML Specification (2.0), Object Management Group (UML Revision Task Force), Sept. 2003, http://www.omg.org/uml.
[5] ITU *Recommendation Z.120: Message sequence chart*. International Telecommunication Union, City, 2000.
[6] Uchitel, S., Brunet, G. and Chechik, M. Synthesis of Partial Behavior Models from Properties and Scenarios. *IEEE Trans. Softw. Eng.*, 35(3)2009).
[7] Whittle, J. and Schumann, J. *Generating statechart designs from scenarios*. City, 2000.
[8] Krka, I., Brun, Y., Edwards, G. and Medvidovic, N. Synthesizing partial component-level behavior models from system specifications. In *Proceedings of the Proc. of the ESEC/FSE* (2009). ACM, [insert City of Publication],[insert 2009 of Publication].
[9] Heninger, K. L. Specifying Software Requirements for Complex Systems: New Techniques and Their Application. *Software Engineering, IEEE Transactions on*, SE-6, 1 1980), 2-13.
[10] Alspaugh, T. A., Faulk, S. R., Britton, K. H., Parker;, R. A., Parnas, D. L. and Shore, J. E. *Software requirements for the A-7E aircraft*. Naval Research Lab, Washington, DC, 1992.
[11] Kripke, S. Semantic considerations on modal logic. *Acta philosophica fennica*1963).
[12] Lee, E. A. *Finite State Machines and Modal Models in Ptolemy II*. Teh.Report UCB/EECS-2009-151, EECS Department, University of California, Berkeley, 2009.
[13] *I. Krüger, Distributed system design with message sequence charts, Dissertation, Technische Universität München, 2000.*
[14] ITU *Recommendation Z.120: Message sequence chart*. International Telecommunication Union, City, 2004.
[15] Benner, K., Feather, M. S., Johnson, W. L. and Zorman, L. A. Utilizing Scenarios in the Software Development Process. In *Proceedings of the Proceedings of the IFIP WG8.1 Working Conference on Information System Development Process* (1993). North-Holland Publishing Co., [insert City of Publication],[insert 1993 of Publication].
[16] Campbell, R. L. WILL THE REAL SCENARIO PLEASE STAND UP? *SIGCHI Bull.*, 24, 2 1992), 6-8.
[17] Campbell, R. L. Categorizing scenarios: a quixotic quest? *SIGCHI Bull.*, 24, 4 1992), 16-17.
[18] Hunter, A. and Nuseibeh, B. Managing inconsistent specifications: reasoning, analysis, and action. *ACM Trans. Softw. Eng. Methodol.*, 7, 4 1998), 335-367.
[19] Uchitel, S., Kramer, J. and Magee, J. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Trans. Softw. Eng. Methodol.*, 13(1)2004).
[20] Uchitel, S. and Chechik, M. Merging partial behavioural models. *SIGSOFT Softw. Eng. Notes*, 29(6)2004).
[21] Alspaugh, T. A., Faulk, S. R., Britton, K. H., Parker;, R. A., Parnas, D. L. and Shore, J. E. *Software requirements for the A-7E aircraft*. Naval Research Lab, Report No. NRL-9194, ,1992.
[22] Parnas, D. L. Designing software for ease of extension and contraction. In *Proceedings of the Proc. of ICSE* (1978), [insert City of Publication],[insert 1978 of Publication].

[23] Forum. *Communications of ACM*, 50(6)2007), 7-9.
[24] Harel, D. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8, 3 1987), 231-274.
[25] Beeck, M. v. d. A Comparison of Statecharts Variants. In *Proceedings of the Proceedings of the 3rd Intl. Symp. Formal Techniques in Real-Time and Fault-Tolerant Systems* (1994). Springer-Verlag, [insert City of Publication],[insert 1994 of Publication].
[26] Mesarovic, M. C. and Takahara, Y. *Abstract Systems Theory*. Springer, 1989.
[27] Courtois, P.-J. and Parnas, D. L. Documentation for safety critical software. In *Proceedings of the Proc. of ICSE* (1993), [insert City of Publication],[insert 1993 of Publication].
[28] Maraninchi, F. and Remond, Y. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Sci. Comput. Program.*, 46(3), 3 2003), 219-254.
[29] Maler, O., Manna, Z. and Pnueli, A. From Timed to Hybrid Systems. In *Proceedings of the Proceedings of the Real-Time: Theory in Practice, REX Workshop* (1992). Springer-Verlag, [insert City of Publication],[insert 1992 of Publication].
[30] Feiler, H. P., Lewis, B., Vestal, S. The SAE Architecture Analysis and Design Language (AADL) Standard. IEEE RTAS Workshop, 2003.
[31] Hirsch, D., Kramer, J., Magee, J. and Uchitel, S. Modes for Software Architectures. In *Proceedings of the EWSA 2006* (2006). LNCS, Springer Verlag., [insert City of Publication],[insert 2006 of Publication].
[32] Foster, H., Uchitel, S., Kramer, J. and Magee, J. *Towards Self-management in Service-Oriented Computing with Modes*. Springer-Verlag, City, 2009.
[33] Jahanian, F. and Mok, A. K. Modechart: A Specification Language for Real-Time Systems. *IEEE Trans. Softw. Eng.*, 20(12)1994).
[34] Paynter, S. *Real-time mode-machines*. City, 1996.
[35] Edmund M. Clarke, J., Grumberg, O. and Peled, D. A. *Model checking*. MIT Press, 1999.
[36] Graf, S. and Saidi, H. *Construction of abstract state graphs with PVS*. City, 1997.
[37] Larsen, K. G. and Thomsen, B. *A modal process logic*. City, 1988.
[38] Parnas, D. L. and Madey, J. Functional documents for computer systems. *Sci. Comput. Program.*, 25, 1 1995), 41-61.
[39] OMG UML Specification (2.0), Object Management Group, Sept. 2003.
[40] Parnas, D. L. Some Theorems We Should Prove. In *Proceedings of the Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications* (1994). Springer-Verlag, [insert City of Publication],[insert 1994 of Publication].
[41] Rushby, J. M. and Srivas, M. K. Using PVS to Prove Some Theorems Of David Parnas. In *Proceedings of the Proceedings of the 6th International Workshop on Higher Order Logic Theorem Proving and its Applications* (1994). Springer-Verlag, [insert City of Publication],[insert 1994 of Publication].
[42] Jing, M. *A Tabel Checking Tool*. Dept. of Elect. and Computer Engineering, McMaster University, 2000.
[43] Sun, J. and Dong, J. S. Design Synthesis from Interaction and State-Based Specifications. *IEEE Trans. Softw. Eng.*, 32(6)2006).
[44] Caso, G. d., Braberman, V., Garbervetsky, D. and Uchitel, S. Validation of contracts using enabledness preserving finite state abstractions. In *Proceedings of*

the Proc. of ICSE (2009). IEEE Computer Society, [insert City of Publication],[insert 2009 of Publication].

[45] Uchitel, S., Kramer, J. and Magee, J. Behaviour model elaboration using partial labelled transition systems. *SIGSOFT Softw. Eng. Notes*, 28(5)2003).

[46] Whittle, J. and Jayaraman, P. K. Synthesizing hierarchical state machines from expressive scenario descriptions. *ACM Trans. Softw. Eng. Methodol.*, 19(3)2010).

[47] Mäkinen, E. and Systä, T. MAS - an interactive synthesizer to support behavioral modelling in UML. In *Proceedings of the Proc. of ICSE* (Toronto, Ontario, Canada, 2001), [insert City of Publication],[insert 2001 of Publication].

[48] Damas, C., Lambeau, B., Dupont, P. and Lamsweerde, A. v. Generating Annotated Behavior Models from End-User Scenarios. *IEEE Trans. Softw. Eng.*, 31(12)2005).

[49] Alrajeh, D., Kramer, J., Russo, A. and Uchitel, S. Learning operational requirements from goal models. In *Proceedings of the Proc. of ICSE* (2009). IEEE Computer Society, [insert City of Publication],[insert 2009 of Publication].