Aspect-Oriented Testing of a Rehabilitation System

Külli Sarna and Jüri Vain
Department of Computer Science
Tallinn University of Technology
Tallinn, Estonia
Kylli.Sarna@eliko.ee; Juri.Vain@ttu.ee

Abstract— **The paper focuses on modularizing test models by adapting aspect-oriented modelling techniques. Model-based testing is an unavoidable part of contemporary model-driven software processes. The essence of model-based testing is to provide methods and tools to validate software systems by generating test cases systematically from models. From the practical usage point of view, it is critical to construct models that capture the essential aspects of the system under test. The proposed test design approach allows systematic separation of testing concerns, that, in turn, helps to overcome the complexity issues. Also, verification conditions are proposed to ensure the correctness of derived aspect test models and their compatibility with base test models. We demonstrate the technique of test model construction using timed automata models and illustrate it with a home rehabilitation system case study.**

Keywords-aspect-oriented testing; model-based testing; test model design; test generation.

I. INTRODUCTION

In the current practice of software testing, including Model-Based Testing (MBT), the test cases are frequently insufficiently structured and specified. The test designers use component-based or hierarchical state models. However, these modelling approaches provide poor support for isolating crosscutting features, specifically, functions that are spread across the software modules and tangled with other functions. We use the principles of Aspect-Oriented Modelling (AOM) to modularize such crosscutting functions into aspects. The AOM approach has evolved from aspect-oriented programming [2] to produce well-structured and well-encapsulated software. We enhance MBT design methodology with aspect handling capabilities taken from AOM [3]. Using the principles of AOM we can encapsulate typical cases like specifying requirements (use cases) that do not specify one property (scattering) or different functionalities (tangling). In this paper, we will explain how to conceptualize concerns into aspects and how to extract test cases from these aspect test models.

In MBT, the tests are generated from formal models of the System Under Test (SUT). The AOM technique introduced by Sarna and Vain [9] models SUT using timed automata and defines aspect models as refinements of the base model. The structural test coverage criteria considered are the same as those commonly used in state models, i.e., state, and transition coverage. As a novelty, in this paper we demonstrate how a test suite can be generated according to structural units that are specific to AOM. This gives us new test coverage criteria that address implemented features – aspect, advice, join-points coverage, etc. - and provide more intuitive reference to the parts of SUT to be tested for those features.

Another advantage of Aspect-Oriented (AO) MBT is the possibility of easy modification of the test suite. When new requirements arise, new advice models can be woven into the test suite without redesigning the existing base model.

Applying the principles of AOM does not provide compositional testing techniques *per se*. Compositionality of proposed AO testing is achieved by imposing extra constraints on how the advice models are constructed and model weaving operations defined. We define these rules in the semantic framework of Uppaal timed automata [6] and formulate the proof obligations to be model-checked. Our approach is illustrated with a home rehabilitation system testing framework.

The rest of the paper is structured as follows. We introduce the technical background in Section 2. Section 3 describes AO MBT. In Section 4, the home rehabilitation system is introduced. Finally, Section 5 concludes the paper.

II. BACKGROUND

A. Aspect-oriented modelling

AOM is a way of modularizing *crosscutting concerns* much like object‒oriented programming is a way of modularizing *common concerns*. *Crosscutting concerns* generally refer to non-functional properties of software, such as security, synchronization, mobility, resilience, etc. In addition, every system may contain its own application specific crosscutting concerns [5].

Cottenier et al. [4] and Rashid [8] have admitted that AOM technologies have the potential to simplify software deployment, and the ability to improve the categorization of crosscutting concerns. Also, AOM aids in modular extension of object systems, where the treatment of crosscutting concerns is encapsulated in separate modules called *aspects*. We use concepts taken from AOM, such as *Aspect, Advice, Join-points, Pointcut*, and *Weaving.*

An aspect consists of two parts: the code/model associated with treatment of the concern (called *advice*), and a predicate defining when the advice should be applied during system executions (called a *pointcut*). The points in the code/model that are identified by a pointcut are called *join-points*.

A *pointcut* selects a subset of join-points based on defined criteria. The criteria can be explicit function names, or function names specified by wildcards. *Pointcuts* can be composed using logical operators. Customized *pointcuts* can be defined, and *pointcuts* can identify *join-points* from different aspects. The process of adding aspects to a base system is called *weaving*; and the result is referred to as the *woven system* [5]. AOM techniques use the term *advice* for the action an aspect will take and *join-points* for where these actions will be inserted in the base system model. *Pointcuts* are used to specify the rules of where to apply an aspect. *Advice*, *join-points*, and *pointcut* are specified as one entity, called an *aspect* [7].

As in AOM, AO testing uses a *base test model* and several *aspect test models.* An example of a base test model is depicted in Figure 1 (for better understanding of the relationship between the models, we use an Automatic Teller Machine (ATM) as an example of a well-known system). The ATM test model specifies the use case of withdrawing money from an ATM. Crosscutting features are treated as patterns described by aspect advice models, and common features are described in the base model. The result of weaving the base model with advice models is called the *composed aspect model*. An advice model can be woven with the base model in many places and in different ways. The Transaction advice model is defined as location refinement of both ATM and Customer automata. The details of advice model construction in the test design level are presented in [9].

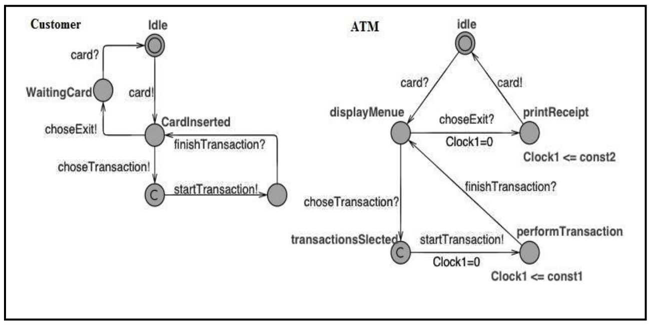

Figure 1. The base test model of ATM.

The base model of an ATM depicted in Figure 1 includes interacting Customer and ATM automata. Refinements in Figure 2 specify aspects of interest: (i) the Transaction advice model is defined as location refinement of both ATM and Customer automata; (ii) edge refinement of ATM. The aspect behaviour is launched from the base model explicitly with the help of channels. We model in Uppaal (www.uppaal.com), a tool box for modelling, simulation and verification of timed automata. In Uppaal [12], the synchronization mechanism is a hand-shaking synchronization: two processes go through a transition at the same time, one will be labelled x !, and the other x ?, where suffixes ?, and ! after the channel name x distinguish sending and receiving synchronization information respectively. A system is composed of concurrent processes, each of them modelled as an automaton. The automaton has a set of locations and edges to specify the control flow. A transition specified by an edge is enabled if its guard and synchronization conditions are satisfied. The transaction automaton in Figure 2 introduces the EnquireBalance aspect

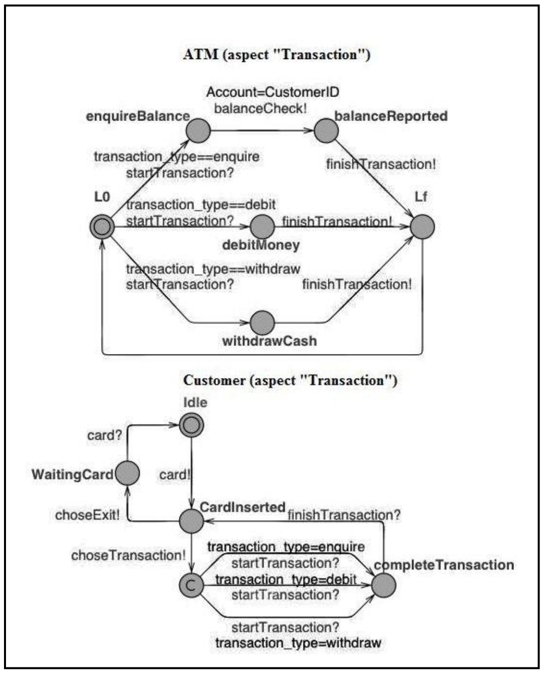

Figure 2. The aspect model “Transaction”.

advice. Since the refinement (ii) introduces a new interaction between ATM and a new actor Server (not shown in the model) the edge introduced is labelled with the ‘balanceCheck!’ channel. When the aspect related tests have to be generated from the composed model of SUT that includes the automata in Figures 1 and 2, we can ignore all the transactions that the aspects of interest do not depend on. For instance, when testing the balanceCheck! transaction between ATM and Server the tester model is extracted from the composition Customer || Customer(Transaction) by algorithm of [1] so that the test sequence <card!, choseTransaction!, transaction_type := enquire, start-Transaction!, wait,[finishTransaction?| timeout >= const1, TESTFAIL], choseExit!, card?, TESTPASS > can be executed.

B. Model-based testing

MBT uses abstract behavioural models for specifying the expected behaviour of the SUT and for automatically generating tests to check if the behaviour of SUT conforms to the model. The SUT is an executable implementation which is considered as a black-box during the testing process, i.e., only inputs and outputs of the system are visible externally. The SUT is tested incrementally by applying test cases. A test case in MBT is defined as a sequence of test stimuli paired with expected SUT outputs. A specified set of test cases constitutes a test suite.

C. Uppaal timed automata

Assume a finite alphabet Σ ranged over by *a*, *b*,... stands for actions and a finite set *C* of real-valued variables ranging over by *x*, *y*, *z*, standing for clocks.

A guard is a conjunctive formula of atomic constraints of the form $x \sim n$ for $c \in C$, $\sim \in \{\geq, \leq, =, >, <\}$ and $n \in$ N. We use $G(C)$ to denote the set of guards, ranged over by g.

Definition 1 (**Timed Automaton**) [6]
A timed automaton ***A*** *is a tuple* $\langle N, l_0, E, I \rangle$ where
– N is a finite set of locations (or nodes),
– $l_0 \in N$ is the initial location,
– $E \in N \times G(C) \times \Sigma \times 2^C \times N$ is the set of edges and
– I: $N \rightarrow G(C)$ assigns invariants to locations (here we restrict to constraints in the form: $x \leq n$ or $x < n$, $n \in$N. For shorthand we write $l \rightarrow_{g,a,r} l'$ to denote $\langle l, g, a, r, l' \rangle \in E$.
To model concurrent systems, timed automata are extended with parallel composition. In the UPPAAL modelling language, the CCS parallel composition operator is used, which allows interleaving of actions as well as hand-shake synchronization. The parallel composition of a set of automata is the product of the automata.
The semantics of timed automata is defined as a transition system where configuration consists of the current location, valuation of state variables and the current values of clocks. There are two types of transitions between states: the automata may either delay for some time (delay transition), or follow an enabled edge (action transition).

To keep track of the changes of clock values, we use functions known as clock assignments mapping C to the non-negative reals R$_+$. Let u, v denote such functions, and $u \in g$ means that clock values denoted by u satisfy the guard g. For $d \in$ R$_+$ let $u + d$ denote the clock assignment that maps all $x \in C$ to $u(x) + d$ and for $r \subseteq C$ let $[r \mapsto 0]$ denote the clock assignment mapping all clocks to 0 and agree with for the other clocks in $C \backslash r$.

Definition 2 (**Operational Semantics**) [6]
The semantics of a timed automaton is a transition system (also known as a timed transition system) where states are pairs $\langle l, u \rangle$ and transitions are defined by the rules:
– $\langle l, u \rangle \rightarrow_d \langle l, u + d \rangle$ if $u \in I(l)$ and $(u + d) \in I(l)$ for a non-negative real $d \in$ R$_+$
- $\langle l, u \rangle \rightarrow_a \langle l', u' \rangle$ if $l \rightarrow_{g,a,r} l'$, $u \in g$, $u' = [r \mapsto 0]u$ and $u' \in I(l')$.
To increase the modeling power keeping the analysis traceable for planner synthesis we lift the model class to rectangular timed automata where guard conditions are in conjunctive form with conjuncts including besides clock constraints also constraints of integer variables.

Similarly to clock conditions, the integer variable conditions are of the form $k \sim n$ for $k \in$ Z, $\sim \in \{\geq, \leq, =, >, <\}$ and $n \in$ N. The advantage of this extension is that the model has rich enough modelling power to represent real-time and resource constraints being same time efficiently decidable for reachability analysis.

III. Aspect-Oriented Model-Based Testing

In this section, we explain the concepts of AOM applicable in aspect-oriented MBT. The AOM allows the models to be organized so that they address particular requirements (including crosscutting ones) and corresponding test cases. The AO test model includes a base model and aspect-related advice models. Aspects may contain sub-aspects that require sub-advices and their own test cases. Sub-aspect models have to be easily inserted into their parent aspect models. In our examples, we use name prefixes that refer to the parent models so that they are convenient to comprehend and maintain.

AO testing can also be considered as an example of compositional testing where the test results of the composed system can be inferred from the test results of its components. In the MBT context, it means that the test cases are determined only by the context of the aspect advice models and the interface behaviour of their composition. AOM also provides a conceptual basis for defining test coverage criteria in terms of aspect related model elements. The hierarchy of those criteria is depicted in Table I.

TABLE I. AO TEST COVERAGE CRITERIA

Coverage constraint / Type of coverage entity	Strong (universal) coverage $\forall$	Weak (existential) coverage $\exists$	Discriminating predicate
Aspect A	All aspects of the model $\forall A \in \boldsymbol{A}.$ ...	Some aspects of the model $\exists A \in \boldsymbol{A}.$ ...	Predicate on aspect constants /variables
i-th join point $jp(A, i)$	All join points of aspect A $\forall jp(A, i) \in JP(A)$	Some join points of aspect A $\exists jp(A, i) \in JP(A)$	Point cut condition
Entry-exit path λ of an advice model $M^{A'}$ $\lambda \in Paths(M^{A'})$	All paths initiated at i-th join point $\forall \lambda \in Paths(M^{A'})$	Some paths initiated at i-th join point $\exists \lambda \in Paths(M^{A'})$	Path predicate, e.g. constraint on path length
Model element of type ***T*** (location, transition, function, data, etc) included in the path $\lambda \in Paths(M^{A'})$	All elements of type ***T*** in $M^{A'}$	Some elements of type ***T*** in $M^{A'}$	Predicate on the attributes of type ***T***

The criteria shown in Table I can be expressed as closed 1st order logic formula in prenex normal form, where the signature includes variables of particular types of structural elements of Uppaal Timed Automata (UPTA) (template, location, transition, label, function, data, etc.). The prefix of the prenex formula includes bound variables in a fixed order that is determined by the natural hierarchy of modelling entities: aspect, join-points, and path. These entities model the structural elements of UPTA, where the structural elements can be referred to directly by name or indirectly by constraints on their attributes. The matrix part may include discriminating predicates of all the above listed types.

The semantics and scoping of AO coverage constraints is defined by the hierarchy and type structure of AO model elements (left most column in Table I). Thus, the scope of constraints on bound variable in the formula matrix part is defined by the position of the bound variable in prefix. For instance, the scope of a path constraint is defined by the join-point and aspect constraints because these elements precede path variable in the prefix. When not explicitly expressed in coverage constraint the default scoping means existential quantification over all those variables preceding in the prefix of coverage constraint. For characterization of coverage criteria in terms of Uppaal query language, we assume that the aspect model *M* is constructed according to the rules described in [9]. The idea is to use Uppaal model checker queries for selecting traces that constitute the test paths of the given test case. Uppaal query based online test generation methods are described by Vain et al. [1] and Hessel et al. [10].

Aspect Coverage criteria impose to execute all or some aspects in a woven model at least once. In *Strong Aspect Coverage* (SAC), given an aspect model *M*, all possible test paths must be covered by the tests. To implement the Strong Aspect Coverage we use the parameterized UPTA templates where the template parameter p_i ranges over indexes [1, *n*] that identify the aspect. Let `P(i)` be a predicate updated to true whenever the i-th aspect advice model is entered. Then the traces of *M* (p_i) under Strong Aspect Coverage criteria should satisfy the query: `E<> forall (i: int [1,n]) P(i)`. Note that given query is valid only for paths that include traversal of all aspects' advice models. In general, the model *M* may not be fully connected and a single path including all aspects may not exist. Therefore, we introduce an auxiliary *reset*- transition into *M* that guarantees that if *n* advice models are reachable in *M* then at most with *n* traversals all of them are visited. The *reset*-transition connects the final location of *M* with its initial location. Due to this construct the Uppaal model checker is able to generate a trace that includes visits of all advice models. The tests paths for a final test case can achieved simply by "cutting" that trace at *reset*- transitions to many shorter sub traces.
Weak Aspect Coverage (WAC) refers to the case where at least one advice model of some aspect is traversed by the test path. The query `E<> forall (i:int [1,n]) P(i)` differs little from the strong coverage constraint but it does not require including *reset*-transitions in the model *M*.

Join Point Coverage criteria impose to execute all or some join points of each aspect in a woven model at least once. *Strong Join Point Coverage* (SJPC) presumes similarly to strong aspect coverage introduction of an auxiliary *reset*-transition into *M*. Regardless the prefix (SAC or WAC) of the query the SJPC contributes a conjunct of form ...`forall (j: int [1,m]) P(i) && R(j)` where `j` is ranging over join point indexes of the aspects referred in the prefix of that query and `R(j)` is a Boolean variable at each join point updated to `true`, whenever this join point is visited. *Weak Join Point Coverage* (WJPC) is satisfied if there is at least one trace for given formula prefix satisfying ...`exists (j: int [1,m]) P(i) && R(j)`. Here, like in WAC, auxiliary *reset*-transition is not needed.

Aspect Path Coverage criteria impose to execute all or some paths of each aspect in a woven model at least once. Assume the entry and exit transitions of each advice models are decorated with *entry*(*i*, *j*,*k*) and *exit*(*i*, *j*,*l*) predicates where *i*, *j*, *k*, *l* range over the set of aspects, join points, and their advice entry and exit points respectively. Whenever the transition is executed these predicates evaluate to `true`. Then, the *Strong Aspect Path Coverage* (SAPC) contributes a conjunct to the query prefixed with aspect and join point constraints as follows: ... `forall (k: int [1,K]) forall (l: int [1,L]) P(i) && R(j) && [(`$\vee_{k=1,K}$` entry(k))` $\wedge$`(`$\vee_{l=1,L}$` exit(l))`. SAPC, like earlier strong coverage criteria, presumes the *reset*-transitions related construct. *Weak Aspect Path Coverage* (WAPC) comparing to SAPC replaces universal quantifiers with existential ones for variables k and l, the coverage constraint becoming to ... `exists(k: int[1,K]) exists(l: int[1,L]) P(i) && R(j) && [(`$\vee_{k=1,K}$` entry(k))` $\wedge$ `(`$\vee_{l=1,L}$` exit(l))`.

The **Model Element Coverage** criteria impose constraints on the types of UPTA elements to be covered in the advice model or set the specific constraints on the attributes of those elements, e.g. *Strong* (resp. *Weak*) *Model Element Coverage* can be parameterized with the element type, e.g. `Transition` and universally (resp. existentially) quantified over given type. More specific coverage constraints can be constructed using type discriminating predicates on, e.g., local data variables of an advice model.

IV. EXAMPLE: TESTING HOME REHABILITATION SYSTEM

The AO MBT approach described in Section 3 has been applied in testing a Home Rehabilitation System (HRS). The model-based testing is needed in the medical domain because of the safety critical nature of the systems and non-trivial combination of functional, performance and security features [11]. The HRS is an application which drives sensor devices, analyses the gathered data, interacts with the patient and submits relevant information to the hospital through the Internet. HRS software contains the following subcomponents: dedicated health hub as communication gateway; vital signals' sensor system for patient measurements; movement tracking sensor system for fall detection, physical activity and exercise monitoring.

There are three actors, namely, Patient, Plan and Sample, interacting in the "home exercising" use case. The composition of automata *Plan* and *Sample* constitute the base model that can be woven with different advice models depending on what body characteristic (pulse, blood pressure, etc.) is monitored. For instance in Figure 3, the use-case *exercising* is refined with two advice models that are instances of the same automaton template. The advice models linked to the base model are location refinements of the unnamed location in the automaton *Sample*. Channel *Sample* ensures that the advice models are executed synchronously with the edge departing from location *Measure* in the automaton *Sample*. A weak join point coverage of completing exercising can be specified now using query `E<>exists(Screen=UB_warning[1])`. The test case ensures that while a patient is exercising, a warning will be shown on a screen when the patient's pulse is greater than the number in U_bound. On the other hand U_bound is the upper value of pulse that the patient may have during exercising and this is specific to each patient. For example if the U_bound is 140 then a warning on a screen goes red and warns "wait until your pulse will be normal". We measure the pulse under "measurement [1]" and an upper bound and a lower bound are indicated. A normal pulse measurement have to be between U_bound and L_bound.

A strong join point coverage of completing exercising can be specified using query E<>`forall (Screen=normal[1])measurement[1]>=L_bound [1]&&measurement[1]<=U_bound[1]`. That means the screen indicates in green that everything is alright and the patient can continue exercising because their pulse is within the allowed range. By this strong join point test coverage, we ensure that our system is able to give the right warnings whenever necessary.

V. Conclusion

In this work, we have introduced an aspect-oriented approach to model-based testing in the context of Uppaal timed automata specifications. We advocate the view that aspect-oriented models help in constructing models of system under test in a systematic and user friendly way, thus helping to defeat the perennial problems of MBT - complexity of construction and maintenance of test models. It has been shown how the aspect related test coverage criteria can be formalized in a systematic way in Uppaal query language Timed Computation Tree Logic (TCTL) and the feasibility of test suites verified on aspect models before real tests are deployed and executed.

Our focus on how a test case can be generated according to structural units that are specific to AOM is novel. This gives new test coverage criteria that address implemented features – aspect, advice, join-points, etc., and provide more intuitive reference to the parts of SUT to be tested for those features.

Another contribution for enhancing MBT by aspects is the possibility of easy update of test case related models. If new requirements arise, new advice models can simply be incorporated by well-defined composition rules. This is especially relevant in regression testing.

Acknowledgment

This research was supported by the European Social Fund's Doctoral Studies and Internationalisation Programme (DoRa), and by the Competence Centre Programme of Enterprise Estonia.

References

[1] J. Vain, M. Kääramees, and M. Markvardt, "Online testing of nondeterministic systems with reactive planning tester," in: L. Petre, K. Sere, and E. Troubtsyna (Eds.). Dependability and Computer Engineering: Concepts for Software-Intensive Systems. Hershey, PA: IGI Global (2012), pp. 113-150.

[2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. - M. Loingtier, and J. Irwin, "Aspect-Oriented Programming", ECOOP'97, June, 1997, pp. 220-242.

[3] J. Kienzle, A. Wisam, F. Fleury, J. Jezequel, and J. Klein, "Aspect-Oriented Design with Reusable Aspect Models." Transactions on Aspect-Oriented Software Development, vol7, 2010, pp. 279-327.

[4] T. Cottenier, A. van den Berg, and T. Elrad, "Stateful Aspects: The Case for Aspect-Oriented Modeling." Proceedings of the 10th AOM Workshop, 2007, pp. 7-14.

[5] E. Katz, and S. Katz, "User queries for specification refinement treating shared aspect join points." Proceedings of the 8th IEEE, 2010, pp. 73-82.

[6] J. Bengtsson, and W. Yi, "Timed automata: Semantics, algorithms and tools." Lecture Notes on Concurrency and Petri Nets, Lecture Notes in Computer Science vol. 3098, 2004, pp. 87-124.

[7] S. Clarke, and E. Baniassad, Aspect-Oriented Analysis and Design: The Theme Approach. Addison-Wesley Professional. 2005.

[8] A. Rashid, "Aspect-Oriented Requirements Engineering: An Introduction". In Proceedings of 16th IEEE, 2008, pp. 173-182.

[9] K. Sarna, and J. Vain, "Exploiting Aspects in Model-Based Testing," in Proceedings of 11th FOAL. ACM, New York, NY, USA, 2012, pp. 45-48.

[10] A. Hessel, K. G. Larsen, P. Pettersson, and A. Skou," Testing Real-Time Systems Using UPPAAL." Lecture Notes in Computer Science vol. 4949. 2008, pp. 77-117.

[11] A. Kuusik, E. Reilent, K. Sarna, and M. Parve, "Home telecare and rehabilitation system with aspect-oriented functional integration." Biomedical Engineering, DOI: 10.1515/bmt-2012-4194 (accessed 01.08.2014).

[12] K. Larsen, P. Pettersson, and W.Yi. UPPAAL in a nutshell. Journal on Software Tools for Technology Transfer, 1997, pp. 134-152.

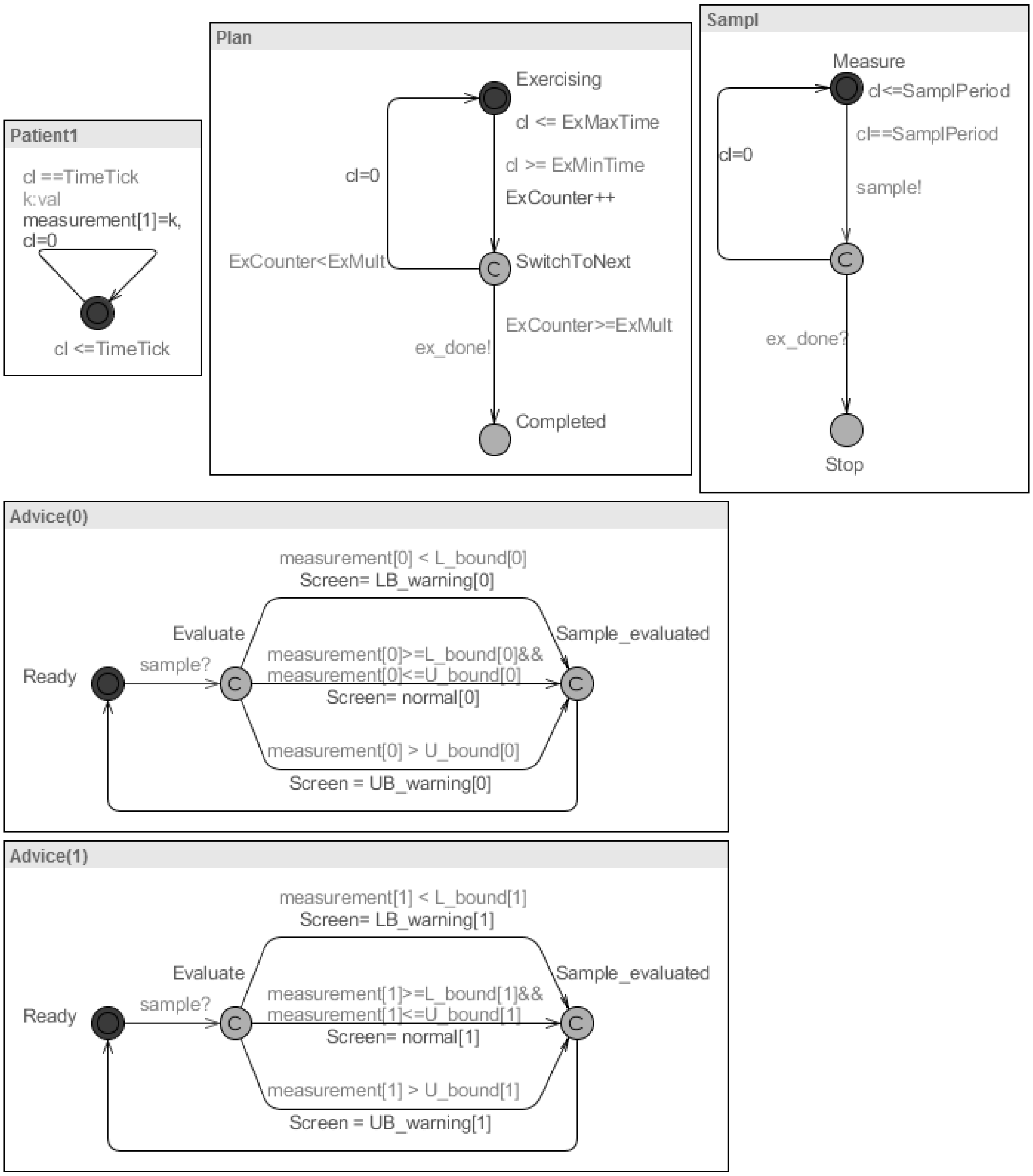

Figure 3. Composing the primary test models and advice model in parallel.