Model Generation by Moderated Regular Extrapolation

Andreas Hagerer[1], Hardi Hungar[1], Oliver Niese[2], and Bernhard Steffen[2]

[1] METAFrame Technologies GmbH, Dortmund, Germany
{AHagerer,HHungar}@METAFrame.de
[2] Chair of Programming Systems, University of Dortmund, Germany
{Oliver.Niese,Steffen}@cs.uni-dortmund.de

Abstract. This paper introduces **regular extrapolation**, a technique that provides descriptions of systems or system aspects a posteriori in a largely automatic way. The descriptions come in the form of models which offer the possibility of mechanically producing system tests, grading test suites and monitoring running systems. Regular extrapolation builds models from observations via techniques from machine learning and finite automata theory. Also expert knowledge about the system enters the model construction in a systematic way. The power of this approach is illustrated in the context of a test environment for telecommunication systems.

1 Motivation

The aim of our work is improving quality control for reactive systems as can be found e.g. in complex telecommunication solutions. A key factor for effective quality control is the availability of a specification of the intended behavior of a system or system component. In current practice, however, only rarely precise and reliable documentation of a system's behavior is produced during its development. Revisions and last minute changes invalidate design sketches, and while systems are updated in the maintenance cycle, often their implementation documentation is not. It is our experience that in the telecommunication area, revision cycle times are extremely short, making the maintenance of specifications unrealistic, and at the same time the short revision cycles necessitates extensive testing effort. All this could be dramatically improved if it were possible to generate and then maintain appropriate reference models steering the testing effort and helping to evaluate the test results.

We propose a new method for model generation, called (moderated) *regular extrapolation*, which is tailored for a posteriori model construction and model updating during the system's lifecycle. The method, which comprises many different theories and techniques, makes formal methods applicable even in situations where no formal specification is available: based on knowledge accumulated from many sources, i.e. observations, test protocols, available specifications and last not least knowledge of experts, an operational model in terms of a state

R.-D. Kutsche and H. Weber (Eds.): FASE 2002, LNCS 2306, pp. 80–95, 2002.

and edge-labeled finite automaton is constructed that uniformly and concisely resembles the input knowledge in a way that allows for further investigation.

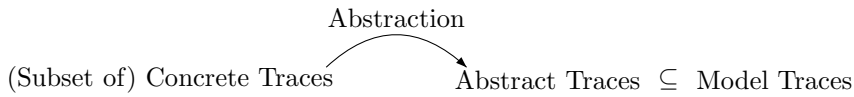
Though it is particularly well suited to be applied in regression testing (cf. section 2.2), where a previous version of the system is available as an approximate reference, regular extrapolation is not limited to this situation. Indeed, our experience so far has been encouraging. We were able to extrapolate an expressive model for some part of a nontrivial telephone switch as a basis for a monitor application (cf. section 5.3), and we could demonstrate its power for test-suite enhancement. Both applications are illustrated in a companion demo paper [8].

The paper is structured as follows: In section 2 we give a short overview about the considered scenario and discuss the design decisions. Section 3 briefly describes the ingredients of our approach for the model generation. The following section provides more details on some of the less standard techniques and the use we make of them regarding the considered scenario: the test of complex telecommunication systems. The usage of the generated models is described on the basis of examples in Section 5. Finally Section 6 draws some conclusions.

2 Regular Extrapolation

2.1 Sketch of the Approach

A key feature of our approach is the largely automatic nature of the extrapolation process. The main source of information is observation of the system, i.e. a set of system traces. These traces may be obtained passively by profiling a running system, or they may be gathered as reactions of the system to external stimulation (like in testing). These traces are abstracted and, after the introduction of a concept of state based on observable system attributes, they are combined into an automaton. This automaton extrapolates from the finite traces which have been observed to infinite behavior following regular patterns. Its language contains all abstract images of the traces observed so far. The general picture is as follows:



Extrapolation from passively obtained observations and protocols of test runs may yield a too rough model of the system, leaving out many of its features and generalizing too freely. So these models have to be refined. We adapt machine learning algorithms and also incorporate expert's knowledge. Learning consists in running specific tests with the aim of distinguishing superficially similar states and finding additional system traces. Experts can either be implementors or people concerned with the environment of the system, for instance people knowing the protocols to be observed. Their knowledge enters the model in the form of declarative specifications, either to rule out certain patterns or to guide state distinguishing searches. Technically this is done by employing the bridge from temporal logic to automata, model-checking techniques and partial-order methods.

Conflicts arising during the extrapolation process between the different sources of information have to be examined and resolved manually (moderation).

2.2 The Regression Testing Scenario

Regression testing provides a particularly fruitful application scenario for regular extrapolation. Here, previous versions of a system are taken as the reference for the validation of future releases: changes typically concern new features, enhanced speed or capacities, some bugs or other often customer-driven change requests. However, by and large, the new version should provide everything the previous version did. I.e., if we compare the new with the old, there should not be too many essential differences. Thus if it were possible to (semi-) automatically maintain models comprising the knowledge obtained during previous development, testing, use and updating phases, regression testing could be largely improved. Besides providing a structure for managing the test suites, these models would be capable of providing flexible means for test run evaluation: note that it is inadequate to simply compare protocols of test runs with stored reference protocols, as besides the few essential differences there are many inessential ones, and it is very hard to distinguish between those two types. However there are formal means to construct models that factor out many of the minor differences between successive versions and thus reduce the manual effort of grading a test run.

2.3 The Design Decisions

Starting point of our investigation was the regression testing problem in the so-called black box scenario: a technique was needed to deal with a large legacy telephony system in which most of the involved applications (the so-called "plus products") running on and with the platform are third party. There was no hope to obtain formal specifications. The only source of information were intuitive user manuals, interaction with experienced test engineers and observations, observations, observations. As none of these sources could be fully trusted (and since what is true today may not be true tomorrow), the only approach was to faithfully and uniformly model all the information and to manually investigate arising inconsistencies. This lead to a change management process with continuous moderated updates:

- initially automata theoretic techniques are used to construct a consistent operational model along the lines of [19] (already here, expert knowledge may be required to guarantee consistency (cf. section 3 and 4)),
- this initial model is immediately used for test evaluation and monitoring (section 5.3).
- whenever unexpected observations arise, these are taken care of by either modifying the model or correcting the running system. Whereas debugging the system is standard, model modification is again done using involved automata-theoretic means (cf. section 3 and 4).

It is impossible in practice to find a precise model of the system under consideration, even on an abstract level. Such a model would usually be far too large and, as results from learning theory indicate, too time-consuming to obtain and to manage. Instead we are aiming at concise, problem-specific models, expressive enough to provide powerful guidance and to enhance the system understanding. In fact, many quite different models of this kind may be required to cover different goals, like monitoring, test generation, test grading or even simulation.

Also, in contrast to "classical" modeling scenarios, we cannot expect our models to *safely* approximate the intended system behaviors, as there is no safe information we can base on. On the one hand, this complicates the moderation in case of discrepancies between the modeled and the observed behaviors. On the other hand, it allows us to use with no loss powerful automata theoretic methods which do not preserve safe approximation. This accelerates the gain of expressiveness of our models, making them a strong aid already very early on.

It is of course very important that the models are reasonably close to the system. Exploiting all the information at hand, independently of their source, to obtain a comprising "hypothesis" model is the best we can do. In fact, our experience with a nontrivial real-life system[1] indicates that "brave" guesses are much better than too conservative ones, as the interaction with the "hypothesis" model enhances the expert's system understanding, and the closer the interaction the faster proceeds the extrapolation process.

2.4 Related Work

Central to our work is the theory of finite automata and regular languages as described for instance in [10]. A less known and more recent part of that theory concerns the problem of determining a model in terms of a deterministic finite automaton from observations. This is intensively discussed in the domain of *machine learning* [13]. There exists in general no efficient method that can compute such a model in polynomial steps. So several methods try through weakening of the requirements or through additional information to achieve the aims. The two most prominent learning models are the *Probably approximately correct learning model (PAC learning)* [24] and the *Query learning model* [1]. Whereas in PAC learning the algorithms gets random positive examples, is it possible in the Query learn model to inquire information about the investigated system actively. To some extent our approach is orthogonal to both of these learning models. The reason for this is basically that we do not aim at determining an exact model for an unknown system, which is unrealistic in practice. Rather we express all the available heterogeneous information about a system (observations, expert knowledge, ...) in a consistent and uniform way. This is similar to the approach of *unifying models* [19] where a heterogeneous specification, consisting of several aspects specified in different formalisms, is combined into a single consistent model.

[1] Gained with a prototype implementation built on top of our already existing *Integrated Test Environment* [16].

3 Ingredients of a Posteriori Model Generation

This section describes the ingredients of a posteriori model generation. The considered class of systems are complex, reactive systems, as can be found e.g. in telecommunication systems. These systems normally consist of several subcomponents which communicate with and affect each other, typically via standardized protocols. As a prerequisite to our approach the system has to provide *points-of-observation (PO)* and *points-of-control-and-observation (PCO)*, i.e. we must be able to make observations of the system and, additionally, to influence it in order to test its reactions to certain (critical) stimuli sequences.

Fig. 1 sketches briefly our iterative approach. It starts with a model (initially empty) and a set of observations. The observations are gathered from a reference system in the form of traces. They can be obtained either *passively*, i.e. a running reference system is observed, or *actively*, i.e. a reference system is stimulated through test cases. The set of traces (i.e. the observations) is then be preprocessed, extrapolated and used to extend the current model. After extension the model is completed through several techniques, including adjusting to expert specifications. The last step validates the current hypothesis for the model, which can lead to new observations.

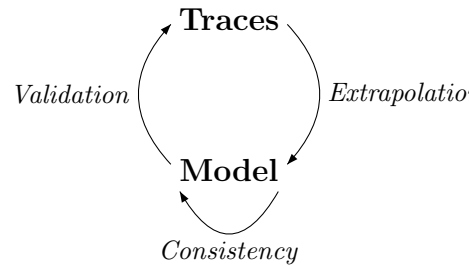


Fig. 1. Generation of models

The realization of this approach uses heavily automata operations and related techniques. The basis is given by standard automata operations like homomorphisms and boolean operations like meet, join and complement [10]. These are complemented by some specific operations for extrapolation. In particular, these are abstraction mappings more general than language homomorphisms, and a particular folding operation, which by identifying similar states introduces cycles in the finite traces. This borrows from automata learning techniques as discussed in [1] but is quite different in several aspects.

The adequate incorporation of expert knowledge requires further techniques. On the one hand, temporal logic [5] serves to formulate specifications which limit the model from above. I.e., experts formulate properties which they believe to be true of the system, and extrapolation results should be limited by them. Temporal-logic model checking is employed to check adherence of the model to these constraints. Counter examples generated by the model checker in case of a violation are to be studied to pinpoint the source of the discrepancy.

On the other hand, experts can identify events in sequences which lead to distinguishing similar states which would otherwise be identified in the extrapolation step. A third way in which expert knowledge enters is in specifying independence relations between events. This, by employing ideas of partial order

approaches [15,25], leads to generalizing from (randomly) sequential observations to parallel patterns.

Finally, the validation of models draws on testing theory (cf. e.g. [14] for a survey) to generate stimuli sequences helping to discover wrongly identified states and missed behavior.

In the following, the steps of the model construction process will be explained one by one. Further details on some of them are given in Section 4.

3.1 Extrapolation

The model will be extended through a set of observations in form of traces. However before the traces are added to the model they have to be generalized. This comprises two steps:

1. abstraction from unnecessary detail, and
2. folding the tree of traces to a finite automaton with joins and cycles.

Abstraction. Abstraction has two aspects: *focus* and (true) *abstraction*. Focus means that if we are not interested in certain events or parameters we can eliminate them from the observation traces. An example are concrete time stamps. (True) abstraction takes care of first-order aspects we cannot simply ignore like participant identities. These have to be represented by propositional means to fit into the world of finite automata. Generally, we restrict the models to represent only instances of behavior with a certain bound on the number of active participants (or other first-order valued sets).

Folding. Before the traces are added to the model we combine them into a single *trace automaton*. If all traces start in the initial state of the system, they are merged using an unique start state into a tree. After that, all states which are seemingly equivalent will be identified. In our telephony system application, states are identified only if all external observations are the same. In particular, each telephone must have the same display and LED state. Further distinguishing criteria formalized as expert knowledge may refer to the history of traces.

3.2 Consistency

Each extrapolation step is followed by a consistency check. It checks whether the extension performed is consistent with the expert specification bounding the permitted behavior. The specifications are given as linear-time temporal-logic (LTL, [5]) constraints. Each constraint defines a formal language, i.e. a set of traces. These constraints are interpreted as loose upper bounds of the system language. The system, and therefore the model, should not contain any trace violating any of the constraints. As the models are finite automata, the LTL constraints can be checked one by one using a model checker. The model checker

either ascertains that a constraint is satisfied or it produces a counter example, consisting of a trace of the model which violates the constraint. The discovery of such errors leads to an investigation whether the specification is wrong, or the extrapolation was too loose, or there is in error in the trace set which lead to the exploration.

This is essentially a manual step to be performed in collaboration with system or application experts. If a constraint is found to be too restrictive, its correction is rather straightforward. Or if it can be attributed to an erroneous observation, i.e. an error in the reference system, its correction is easy: we leave out the observation for the construction of our model and as a side benefit of model construction we have discovered an error. More difficult for our construction procedure is an error introduced by the abstraction step. The simplest, not always appropriate remedy is to remove all paths which violate the constraint. This works for safety constraints, for liveness constraints a deeper analysis is required which removes incorrect cycles.

Besides constraint checking, the consistency check uses the independency relation to complete the model (in the sense of partial-order methods). This is described in more detail in Section 4.4.

3.3 Validation

To ensure the validity of the obtained model a validation step completes the cycle. Here, tests are generated to further check for the correctness of state identifications and to look for additional model traces. I.e., like in learning an automaton from its external behavior, it is tried to verify the current hypothesis against the reference system. Besides state splits necessary to remedy too optimistic distinctions, these tests may lead to new observations which reenter the cycle.

The main point of validation is to make sure that the model is rather precise on short sequences. As most errors will already show up in short sequences,[2] this is not only the easiest but also the most useful thing to do. Remember that we are not aiming at a precise model, but only at an approximation comprising all the information currently available. Thus we do not suffer from the problem of deep "combination locks" [13] which make the learning problem inherently difficult (in the worst case).

Summarizing, tests covering all the essential short sequences are produced in order to validate the models state identifications. I.e. stimuli are generated to observe system reactions on input events which have not been seen so far. This in order to check whether after following different paths in the model which lead to the same state of the model there is indeed no (easily) discernable difference in system behavior. In particular, this is applicable to cycles (cycle validation).

[2] A fact explaining why the current practice does in fact find the most severe errors. Of course, the more complex the overall scenario, the longer are the required "short" sequences. This explains today's urgent need for new test technology.

4 Handling a Posteriori Model Generation

In this section we present the structure of the generated models, the considered application scenario and afterwards details of the key aspects of the extrapolation step and the consistency check.

4.1 Model Structure

Fig. 2 shows a part of a model. The filled states are called *stable system states*, i.e. states in which the system cannot continue without a stimulus from its environment. The other states are called *internal states*. In this example the system is in state S_1 and it can receive e.g. the event (or action) *?a*[3]. The system produces two events (*!x* and *!y*), which are sent to the environment and afterwards it reaches the stable state S_2. In general there are additional observations attached to both states and actions of the model. Examples for state observations are e.g. the display status of a specific device. Action observations can be e.g. concrete parameter values of a protocol event.

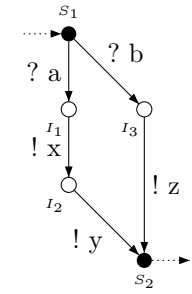


Fig. 2. Model structure

4.2 Application Scenario: System-Level Testing of CTI Systems

Fig. 3 shows the considered scenario, a complex *Computer telephony integrated (CTI) system*, concretely a *Call Center solution*. A midrange telephone switch is connected to the ISDN telephone network or, more generally, to the public switched telephone network (PSTN), and acts as a 'normal' telephone switch to the phones. Additionally, it communicates directly via a LAN or indirectly via an application server with CTI applications that are executed on PCs. Like the phones, CTI applications are active components: they may stimulate the switch (e.g. initiate calls), and they also react to stimuli sent by the switch (e.g. notify incoming calls). Moreover the applications can be seen as logical devices, which can not only be used as a phone, but form a compound with physical devices.

In our point of view the switch has one *PCO* (via the *Corporate Network Protocol* (CorNet)) and one *PO* (via the *Computer Supported Telecommunications Applications Protocol* (CSTA) [4]). The technical realization of these test interfaces is provided by the *Integrated Test Environment* (ITE) [16,18], in particular the *Test Coordinator*. The ITE is an environment for the management of the overall test process for complex systems, i.e. specification of tests, execution of tests and analysis of test runs. The ITE is able to steer different test tools (dedicated to the subcomponents of a complex system) and to coordinate and evaluate the test runs. In the considered scenario two different kind of test tools are used by the test coordinator:

[3] We mutate the *?a* and *!a* notation of process algebra to denote inputs and output actions respectively.

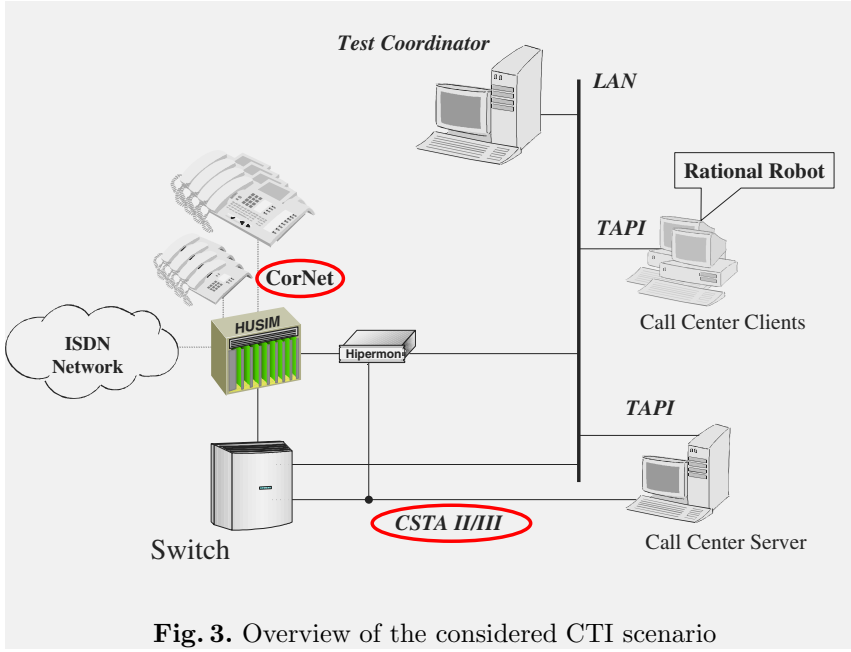


Fig. 3. Overview of the considered CTI scenario

1. A proprietary protocol analyzer (*Hipermon* [9]) which is connected to a telephone simulator (*Husim*) and to the connection between the switch and the application server.
2. A GUI test tool (*Rational Robot* [11]), which is used in several instances, i.e. for every considered call center client.

The heart of the environment is the Test Coordinator tool (cf. fig. 3), built on top of METAFrame's *Agent Building Center* [21], which is a generic and flexible workflow management system. So far the ITE has been successfully applied to the system level testing of complex system solutions like *Web-based applications* [17] or *CTI-Solutions* [16].

Based on this environment we are able to build a model for the telephone switch on CSTA-level [4].

4.3 Performing Regular Extrapolation

Abstraction. Abstraction is concerned with single traces of the system to be modeled. In a first step we filter and generalize the observations attached to states and transitions. In the considered scenario we concretely:

- restrict of the set of devices to be considered
- restrict of the set of physical attributes of system devices which enter the model, e.g.:
  (*ignore: (obs-id, obs-attribute, attribute-value) = (LED, LED-id, 27)*),
- generalize nongeneric information, e.g. date and time in displays are substituted by generic place holders
  (*replace: (obs-id, obs-attribute, attribute-value, attribute-pattern, replacement) = (DISPLAY, DISPLAY-no, 1, "LETTER (2:3) DIGIT (1:2) '.' LETTER (3) DIGIT (2)", "DATE")*)

In a second step concrete identifications of devices and system components are substituted by symbolic names that reflects the "roles" in a telecommunication scenario. Using symbolic names allows identifying information that is partially already included in a model when newly observed system behavior is to be added to the model. This results in smaller models. E.g. different tests that are related to specific switch features may differ only in the activities after establishment of a connection between two devices and in the identifications of the devices concretely used in the tests. Then, determining a common prefix of traces is facilitated if symbolic names are used. Furthermore, models generalized in this way are independent of the environment used to collect the traces. The generalization resembles that we determine symbolic names of actors. An actor encompasses a group of devices and components that have any interrelated association with the control and observation of calls, e.g. displays, buttons and lamps of a telephone set. For each possible actor able to act in the system a specification is prepared which consists of a set of replace-criterion-value pairs and which is used to determine actors associated with observations.

The trace automaton is then further generalized through the *folding step.*

Folding. Behaviorally equivalent stable states are identified through a comparison of their observations and can then be merged. More precisely, two states are (locally) behaviorally equivalent if their attached observations are identical, i.e. concretely all observed devices have the same status regarding display messages and LED's. It is in this step that the behavior of the system is extrapolated: with folding we can obtain an automaton with infinite behavior when cycles are introduced, cf. Fig. 4. However this step provides only an optimistic hypothesis and not a "real" (global) behavioral equivalence, as we cannot ensure that no "hidden" causalities exists. Thus sometimes it is necessary to refine this approach by distinguishing locally similar states, which can elegantly and systematically be done following the property-oriented expansion approach proposed in [20].

4.4 Adding Expert Knowledge

Generalization through Partial Order Methods. Another formalism to specify expert knowledge is inspired from the *partial order reduction* methods for communicating processes [15,25]. Normally these methods help avoiding having

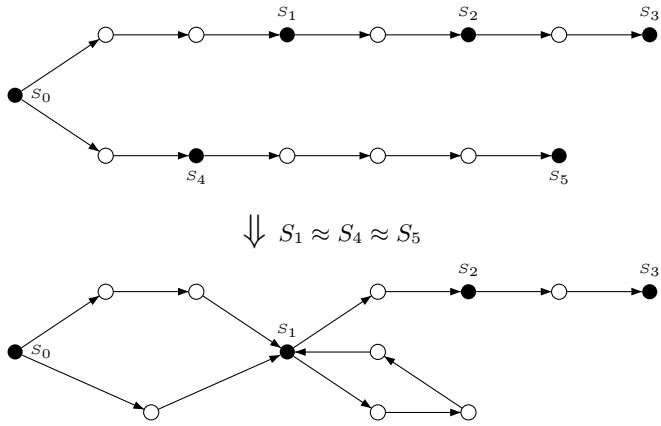


Fig. 4. Folding of equivalent states

to examine all possible interleavings among processes. However these methods can also be used as a specification formalism for generalization in the following way:

1. An expert specifies explicitly an **independence relation**, e.g. *Two independent calls can be shuffled in any order.*
2. A trace is inspected if it contains independent subparts.
3. Instead of the explicit observed trace, a whole equivalence class of traces (of which the observed trace is a representative) can be added to the model.

Partial order methods can be used both for a generalization and for a reduction of the generated models, where the independence relations can be found on a coarse granular level and on a fine granular level.

Coarse Granular Level. Fig. 5 shows how a coarse granular independence relation between stable states can be used for a further generalization of the model. When actions are independent from each other, then it is possible to change the order. The example fig. 5(a) shows two independent *offHook/onHook* sequences: the permutations fig. 5(b) and fig. 5(c) are equivalent to fig. 5(a) from the behavioral point of view.

Fine Granular Level. Under certain circumstances it is however sensible if not necessary to distinguish reorderings between internal states, e.g.: after sending an *?offHook* to the phone the switch responds with at least two messages. One message ensures that the text on the display is updated and the other one sets the actual LED status. If the switch works under normal conditions there exist priorities that schedule the order of the messages. However if the test is done

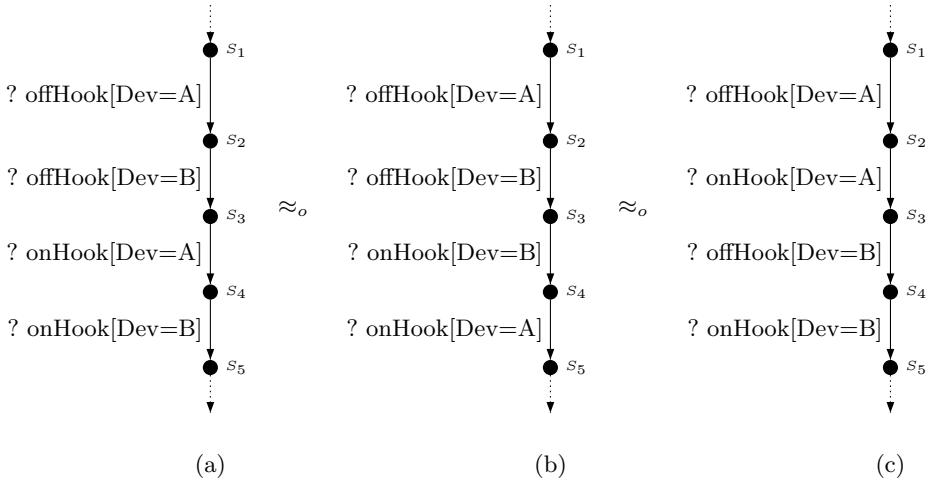


Fig. 5. Examples for reordering

e.g. under performance pressures it is allowed that the messages be reordered, so that other orderings of the events are acceptable and should not be marked as erroneous.

5 Examples for the Usage of Models

Within this section we will demonstrate several applications of the generated models, that enhance the overall test process.

5.1 Enhanced Test Result Evaluation

Current test practice shows that test case evaluation is mostly done via only few observation criteria and is therefore grossly incomplete. The example in Fig. 6(left) shows a typical test case where two devices get connected. The evaluation of the test case is done by observation of the display of the participating devices, which is sufficient for a correct system. However an incorrect system might produce non-conforming display messages or other erroneous signals on one of the other external interfaces, e.g. it is possible that one or more LED's are enabled which should be disabled. Additionally it is also possible that wrong devices were addressed. For our regression test application we therefore gather *all* the visible information of the reference system in response to the test inputs, which provides us with valuable information during the test run/evaluation. Fig. 6(right) shows a fraction of a model with a set of observations. One can see that for *all considered devices* (not only the stimulated ones) the whole set of observations is stored. In particular, this approach allows test engineers to design test cases concisely without bothering about completeness issues.

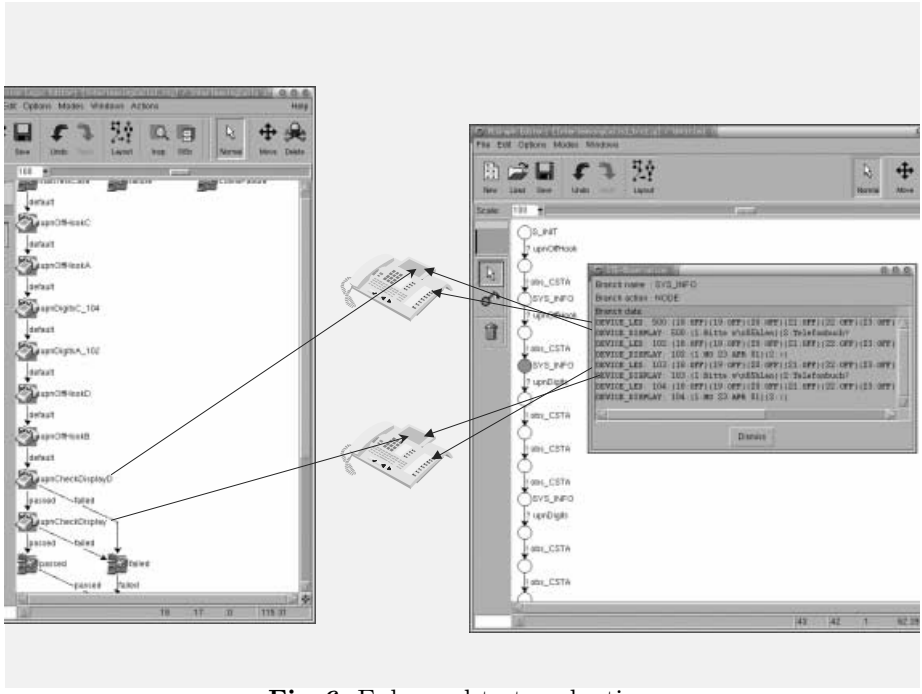


Fig. 6. Enhanced test evaluation

5.2 Improved Error Diagnosis

Automatically refined models in the regression test scenario enable an improved error diagnosis. When a test run evaluates to *failed* the detailed error diagnosis is sometimes a hard task. This is because some causalities between stimuli and events are not straightforward so that stimuli may result in errors not directly observable as erroneous user feedback (e.g. as a display message on a device) but in internal errors that become visible only in later contexts. The error diagnosis for this kind of errors can be quite tricky for a test engineer as he must know about these causalities in order to understand the real reason for errors, which lie (far away) in the past. However in the automatically refined models these causalities are stored and can be used for a more precise error diagnosis. Consider the example of fig. 7, where a trace is shown on the left and the corresponding fraction of the (correct) model on the right. When a test run is evaluated to *passed*, because e.g. $S_2 \approx S_2'$, the missing event must be marked as suspicious and presented to the user as it is possible that it is responsible for later errors. Otherwise, when the test run is evaluated to *failed*, the missing event is probably responsible for the failure and must be object of further investigation by an expert. However it is very useful for an expert to obtain exact hints where (remarkable) differences between the trace and the model lie instead of having to analyze the whole log file.

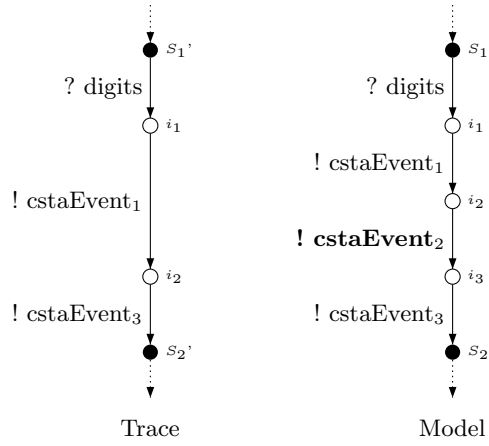


Fig. 7. Examples for Improved Error Diagnosis

5.3 Further Applications

Out of the manifold usages of the generated model, we mention here the most immediately test related ones.

Test suite evaluation and test generation. As soon as a model is generated, all the standard methods for test generation and test evaluation become applicable (e.g. [3,7] for finite state machines or [2,23,6,22] for labeled transition systems resp. input/output transition systems): it can be investigated how much of the model is actually covered by a given test suite, and it is also possible to automatically generate test suites guaranteeing certain coverage criteria. However, in contrast to the classical approach, where the model (specification) is typically considered correct, we do not have such a point of reference: neither the system nor the model can be trusted. Thus it is not clear how well the coverage measured on the model applies to the structure of the real system.

We do not consider this situation as disadvantageous, as in reality, up-to-date models or specifications are very rare, and therefore methods dealing with a symmetric situation are required, where the truth lies "between" the system and specification (model). Moreover, even if up-to-date models exist, our approach is adequate for keeping system and specification aligned.

Monitor. A *monitor* application observes a running system and is able to notify when an error occurs or an unexpected situation arises. In the latter case it is e.g. possible to start automatically a detailed trace. This is particularly useful to catch sporadic errors in systems already delivered to customers. Otherwise, they are hard to detect because it is not feasible to trace over long periods of time. The monitor enables selective tracing, and even a preliminary (and therefore incomplete) model could very well prove to be useful.

6 Conclusion

We have presented an approach, which we called *regular extrapolation*, that solves the specification problem for reactive systems a posteriori. Regular extrapolation comprises many different theories and techniques that make formal methods applicable even in situations where no formal specification is available. Though it is particularly well suited to be applied in regression testing, where a previous version of the system is available as an approximate reference, regular extrapolation is not limited to that situation. Indeed, our experience so far has been encouraging. We were able to extrapolate an expressive model for some part of a nontrivial telephone switch as a basis for a monitor application (cf. section 5.3), and we were able to demonstrate its usefulness for test-suite enhancement, as presented in a companion demo paper [8].

The approximative models resulting from our process are typically no safe approximations of the real systems. In learning complex systems, safety comes at the price of rather inaccurate models. Striving for a best match of all available information accelerates the convergence of the process. However our models can only be used in cases where safety is not a vital requirement. This is true for the majority of scenarios where no formal models exists. There gaining understanding of the system is the major goal. In these cases a thermometer with a precision of 1 percent is preferred to a thermometer with a precision of 5 percent which additional preserves *underapproximation*.

Currently, we investigate various directions to enhance our approach. Particularly important are new methods for diagnosis in case a discrepancy between a system and its model is found. These methods must be more advanced than in the usual safe approximation settings, because no direction of approximation is known. Thus decision support in form of additional diagnostic information is required, in order to decide for adequate updating steps on the model or corrections of the system. Although we are only at the beginning here, we are convinced that our regular extrapolation approach will significantly enhance the impact of formal methods in the industrial practice.

References

1. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 2(75):87–106, 1987.
2. E. Brinksma. A theory for the derivation of tests. *Proc. of PSTV VIII*, pages 63–74, 1988.
3. T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978.
4. European Computer Manufactures Association (ECMA). Services for computer supported telecommunications applications (CSTA) phase II/III, 1994/1998.
5. E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of theoretical computer science*. Elsevier, 1990.
6. J.C. Fernandez, C. Jard, T. Jéron, L. Nedelka, C. Viho. Using on-the-fly verification techniques for the generation of test suites. In *Proc. CAV 1996*, LNCS 1102. Springer Verlag, 1996

7. S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Trans. on Software Engineering*, 17(6):591–603, 1991.
8. A. Hagerer, H. Hungar, T. Margaria, O. Niese, B. Steffen, and H.-D. Ide. Demonstration of an operational procedure for the model-based testing of CTI systems. In *Proc. of the 5th Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2002)*, this Volume.
9. Herakom GmbH. `http://www.herakom.de`.
10. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
11. Rational Inc. The rational robot.
12. B. Jonsson, T. Margaria, G. Naeser, J. Nyström, and B. Steffen. Incremental requirement specification for evolving systems. Nordic Journal of Computing, vol. 8(1):65, Also in *Proc. of Feature Interactions in Telecommunications and Software Systems 2000*, 2001.
13. M.J. Kearns and U.V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.
14. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proc. of the IEEE*, volume 84, pages 1090–1123, 1996.
15. A. Mazurkiewicz. Trace theory. *Petri Nets, Applications and Relationship to other Models of Concurrency*, LNCS 255, pages 279–324. Springer Verlag, 1987.
16. O. Niese, T. Margaria, A. Hagerer, M. Nagelmann, B. Steffen, G. Brune, and H. Ide. An automated testing environment for CTI systems using concepts for specification and verification of workflows. *Annual Review of Communication*, 54, 2000.
17. O. Niese, T. Margaria, and B. Steffen. Automated functional testing of web-based applications. In *Proc. QWE 2001*, 2001.
18. O. Niese, B. Steffen, T. Margaria, A. Hagerer, G. Brune, and H. Ide. Library-based design and consistency checks of system-level industrial test cases. In H. Hußmann, editor, *Proc. FASE 2001*, LNCS 2029, pages 233–248. Springer Verlag, 2001.
19. B. Steffen. Unifying models. In R. Reischuk and M. Morvan, editors, *Proc. STACS'97*, LNCS 1200, pages 1–20. Springer Verlag, 1997.
20. B. Steffen. Property oriented expansion. In *Proc. Int. Static Analysis Symposium (SAS'96)*, LNCS 1145, pages 22–41. Springer Verlag, 996.
21. B. Steffen and T. Margaria. *METAFrame in Practice: Design of Intelligent Network Services*, LNCS 1710, pages 390–415. Springer Verlag, 1999.
22. Q.M. Tan and A. Petrenko. Test generation for specifications modeled by input/output automata. In *In Proc. Of 11th IFIP Workshop on Testing of Communicating Systems (IWTCS'98)*, pages 83–99, 1998.
23. J. Tretmans. Test generation with inputs, outputs, and quiescence. In *Proc. TACAS'96*, LNCS 1055, pages 127–146. Springer Verlag, 1996.
24. L.G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.
25. A. Valmari. On-the-fly verification with stubborn sets. In *Proc. CAV 1993*, LNCS 697, pages 397–408. Springer Verlag, 1993.