TEST-BASED MODEL GENERATION FOR LEGACY SYSTEMS

Hardi Hungar[1,2], Tiziana Margaria[1,2], Bernhard Steffen[1]

[1] University of Dortmund, Dortmund (Germany)
[2] METAFrame Technologies GmbH, Dortmund (Germany)

Abstract

We study the extension of applicability of system-level testing techniques to the construction of a consistent model *of (legacy) systems under test, which are seen as black boxes. We gather observations via an automated test environment and systematically extend available test suites according to learning procedures. Testing plays two roles here: (i) as an application domain and (ii) as the enabling technology for the adopted learning technique. The benefits include enhanced error detection and diagnosis, both during the testing phase and the online test of deployed systems at customer sites.*

1 Introduction

Modern telecommunication and IP-based applications are multi-tiered, distributed applications that typically run on heterogeneous platforms. Their correct operation depends increasingly on the interoperability of single software modules, rather than solely on their intrinsic algorithmic correctness. A scalable, integrated test methodology capable of granting adequate coverage of functional testing, yet still easy to use, has been presented in [11]. This methodology bases on our previous work [6, 14, 10] on system level test of Computer Telephony Integrated (CTI) and Web-based applications, which has been used, for instance, in industrial system-level testing of over 200 COTS applications that interwork with a family of midrange telecommunication switches.

The main immediate benefits of the approach concerned automation of test execution (with a productivity gain of a factor of over 40), structuring, design support, and early validation of test cases, and a much improved analysis of test execution. Not addressed was the design of test suites: all the tests were manually constructed purely on the basis of experience. This was due to the absence of any form of (formal or semi-formal) *model* for the systems that compose a CTI scenario, which are seen and treated as black boxes. In particular, there was no basis for test coverage considerations or focussed test suite enhancement.

In this paper, we propose a pragmatic yet theoretically well-founded and systematic way of constructing expressive hypothesis models, in order to improve test-case evaluation and to support test-suite generation and online testing.

Central idea behind the model construction is to exploit the above mentioned testing machinery as a basis for constructing models by means of automata learning. By steering the learning algorithm appropriately, it is possible to focus the model construction in a way that yields concise models expressing the essential behavioral information, while they at the same time permit automatic analysis and modification techniques. In this approach, testing plays two roles:

- it constitutes the *application domain*: we systematically open testing, and in particular regression testing, online testing and test generation, to systems without a model.
- it constitutes the *enabling technology* for model construction: our active learning technique strongly depends on a so-called membership oracle for determining whether a system may show a certain behavior. This oracle can be emulated by means our automated test equipment.

Our approach will be illustrated in its application to model generation within a commercial call center scenario. We are convinced that this approach has the potential to open enhanced testing methods to legacy systems, which are in practice the large majority.

In the following, we first present the system-level setting, by means of a concrete application (a commercial call-center scenario) in Sect. 2, then we describe our integrated test environment (*ITE*) in Sect. 3 and the conceptual extensions needed for model construction in the main Section 4. Subsequently, the use of models for online testing is treated in Sect. 5, before we

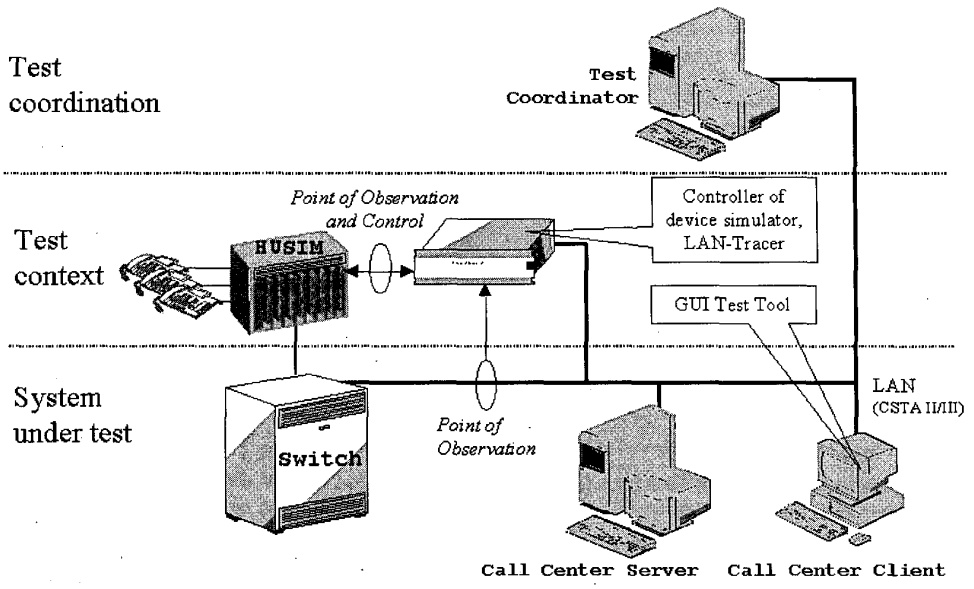

Figure 1: Architecture of the Test Setting for the CTI Application

present our results in Sect. 6 and our conclusions in Sect. 7.

2 The Call Center Application

The general architecture of the considered scenario is illustrated in Fig. 1, which also shows our automated test equipment (ATE) consisting of tools like Rational Robot [15], the Siemens hardware environment simulator HUSIM, and the Hipermon [7] ISDN and IP tracing tool. The system's main part is a midrange telephone switch which is connected to the ISDN telephone network and acts as a 'normal' telephone switch to the phones. Additionally, the switch communicates directly via a LAN or indirectly via an application server with *computer-telephone integrated* (CTI) applications that are executed on PCs.

Here in essence two communication protocols are used: *CorNet* for the communication between the switch and its peripheral devices (phones), and *CSTA/TAPI* for the communication between the switch and the CTI client/server applications that communicate with the switch over a LAN. Like the phones, the 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). The Call Center application runs on the application server, which also runs a *TAPI service provider* that performs a mapping of the TAPI protocol [13] (on which the application and services base) to the *CSTA Phase II/III* protocol [3, 4], an application layer protocol that is transported via TCP/IP.

3 Our ATE: The Integrated Test Environment

To test complex cooperating systems of this kind we add automated test equipment to the System Under Test (SUT). The necessary interface to the SUT is provided by the *Integrated Test Environment ITE*. The core of the ITE, the *test coordinator*, is an independent system that drives the generation, execution, evaluation and management of the system-level tests. In general, it has access to all the involved subsystems and can manage the test execution through a coordination of different, heterogeneous test tools. These test tools, which locally monitor and steer the behavior of the software on the different clients/servers, are technically treated just as additional units under test. The ITE has been successfully applied along real-life examples of IP-based and telecommunication-based solutions: the test of a web-based application (the Online Conference Service, used e.g. for the support of the program committee operations of four ETAPS 2003 conferences [9]) and the test of an IP-based telephony scenario (e.g. Siemens' testing of the Deutsche Telekom's Personal

Call Manager application [10], which supports among other features the role based, web-based reconfiguration of virtual switches). In both cases, no fine grained formal model of any subsystem was available, so that the systems were actually black boxes at the user's level, but a rich collection of expressive test cases, adequate for applying our model generation methodology, was available already a short time after the adoption of the ITE test environment.

In our experiments we derived several models of the switch with the help of this ATE.

4 Using Tests as Observations of Distributed Systems

Being able to systematically test a black-box system constitutes the technical basis for the construction of a valid model. Further ingredients are the introduction of an adequate, abstract view on the system and the organization of the information gathering process to arrive at a valid, useful system representation.

4.1 Abstraction

We use abstraction to reduce the models we have to construct to a manageable size: our main concern is to achieve expressive yet manageable models, retaining the essential information about the system while on the other hand they permit automatic analysis and modification. We chose to model on a *propositional level*, including relevant *control aspects* and *causal dependencies* of the system at hand into the model while leaving out data and real-time aspects. This works particularly well for telephony systems, whose behavior usually does not depend on *what* is transmitted, but on *how*, and which are rather loose in their timing constraints.

Let us consider the communication at the CSTA protocol level. This happens via CSTA records, and as they occur in the communications of our example systems, they convey very rich and detailed information. Some of the record's fields convey essential information relevant to the modelling, while others can safely be ignored. For most modelling purposes, it will be sufficient to project such records to something as abstract as, for instance,

```
(hookswitchOnHook,500)
```

where `hookswitchOnHook` denotes the event that a telephone receiver has been hung up, and `500` identifies the phone.

This 'hiding-based' reduction is the first step towards a propositional view of a system's behavior. However, more involved abstraction techniques are required for our approach to work in the considered scenario.

Small Instantiations A serious problem to attack with our approach is the sheer size of the models. Usually, the number of components connected to a system like the telephone switch might be finite, but rather large. Modelling a system with the maximal number of components (if known) would be unmanageable. Also, a new release might increase this parameter, thus invalidating the model. And last but not least such a large model would not reveal much additional information about the system. In fact, both protocol specifications and practical tests usually work with small, finite instantiations of a system environment.

We do the same, and explore the (full) system only according to a *projective view*: the test cases produced during the learning phase only address a small number of components, just enough to cover the potential interactions. This results in models of manageable size, which, as will be discussed in Section 5.3, can be favorably used for the online testing of the overall system.

4.2 Constructing Models by Learning

Techniques from the field of *automata learning* were our guideline when implementing our model-construction approach. Generally speaking, automata learning tries to construct an automaton that matches the behavior of a given target automaton based on observations of the target automaton and perhaps some further information on its internal structure. We will not discuss the theory of learning here in depth (the interested reader may refer to [5, 16, 8] for our view on learning). Instead, we only present the basic aspects of our realization, which is based on an adaptation of the learning algorithm L* from [1].

L* learns a finite automaton by posing *membership* and *equivalence* queries: a membership query tests whether a string (a potential run) is contained in the target automaton's language, and an equivalence query compares the hypothesis automaton with the target automaton for language equivalence in order to determine whether the learning procedure was (already) successful.

In its basic form, L* starts with the one state hypothesis automaton that treats all words over the considered alphabet (of elementary observations) alike and refines this automaton on the basis of the query results

iterating two steps. Here, the dual way how L* characterizes (and distinguishes) states is central:

- from *below*, by words reaching them. This characterization is too fine, as different words may well lead to the same state.
- from *above*, by their future behavior wrt. a dynamically increasing set of words. These future behaviors are essentially bit vectors, where a '1' means that the corresponding word of the set is guaranteed to lead to an accepting state and a '0' captures the complement. This characterization is typically too coarse, as the considered sets of words are typically rather small.

The second characterization directly defines the hypothesis automata: each occurring bit vector corresponds to one state in the hypothesis automaton.

The initial hypothesis automation is characterized by the outcome of the membership query for the empty observation. Thus it accepts any word in case the empty word is in the language, and no state otherwise. Now the learning procedure 1) iteratively establishes local consistency after which it 2) checks for global consistency:

Local Consistency: This first step (also referred to as automatic *model completion*) again iterates two phases, one checking consistency according to the bit vectors characterizing the future behavior as explained above (this typically leads to state splittings whenever the set of characterizing words increases), and one for checking wether the constructed automaton is closed under the one-step transitions, i.e., each transition from each state of the hypothesis automaton ends in a well defined state of this very automaton.

Global Equivalence: After local consistency has been established, an equivalence query checks, whether the languages of the hypothesis automaton coincides with the language of the target automaton. If this is true, learning successfully terminates. Otherwise the membership query returns a counterexample, i.e., a word which distinguishes the hypothesis and the target automaton. This counterexample gives rise to a new cycle of modifying the hypothesis automaton and starting the next iteration.

In any practical attempt of learning legacy systems, equivalence tests can only be approximated (what we will not explain here), but membership queries can be answered by testing [5, 16].

To apply this technique to our scenario, we first note that the abstraction techniques described above permit us to view the system under test as a finite automaton: we take a finite instance of the system and apply abstraction to all the observed runs. A basis for the model to be constructed can be obtained via the available test runs. The mechanisms of L* can be used to extend this information to an approximate hypothesis model.

L* uses membership queries extensively to systematically explore the system's behavior, until no more 'direct evidence' for inconsistencies between an updated model and system behavior is found.

We rely on being able to answer membership queries with our testing approach. This is not quite as easy as it sounds, simply because a sequence to be tested is an abstract, propositional string, and the system on the other hand is a physical entity whose interface follows a real-time protocol for the exchange of digital (non-propositional) data. Thus we have to drive the system with real data, which requires reversing the abstractions and producing a concrete stimulation string.

In practice, the inversion of abstraction is a concretization function. Things (fields and data) abstracted from the observations have to be filled in dynamically, taking the reactions of the system and basic consistency properties into account. For instance, time stamps have to increase, and instead of symbolic addresses and symbolic tags their concrete counterparts have to be used consistently. Finally, these data have to be transformed into signals and fed to the system.

All this is done by the *ITE*, which was extended in some parts to perform these functions. It performs this task using (1) specific hardware like the Hipermon and Husim, as well as predefined code blocks for generating stimuli and for capturing responses and (2) glue code, which together solve the problems connected with generating, checking and identifying the non-propositional protocol elements. Thus much of the work of putting our approach into practice relies on the *ITE* system and its diverse components.

Fig. 2 and Fig. 3 show a step of the construction process of a model for the switch of our call-center application. In Fig. 2, a partial model on the left and a new trace are shown, which are then combined into a new (still partial) model in Fig. 3. This example shows how linear traces are folded into state graphs of deterministic finite state automata.

5 Scenarios for Practical Application

The outcome of the learning process is a finite, abstract representation of part of the system behavior.

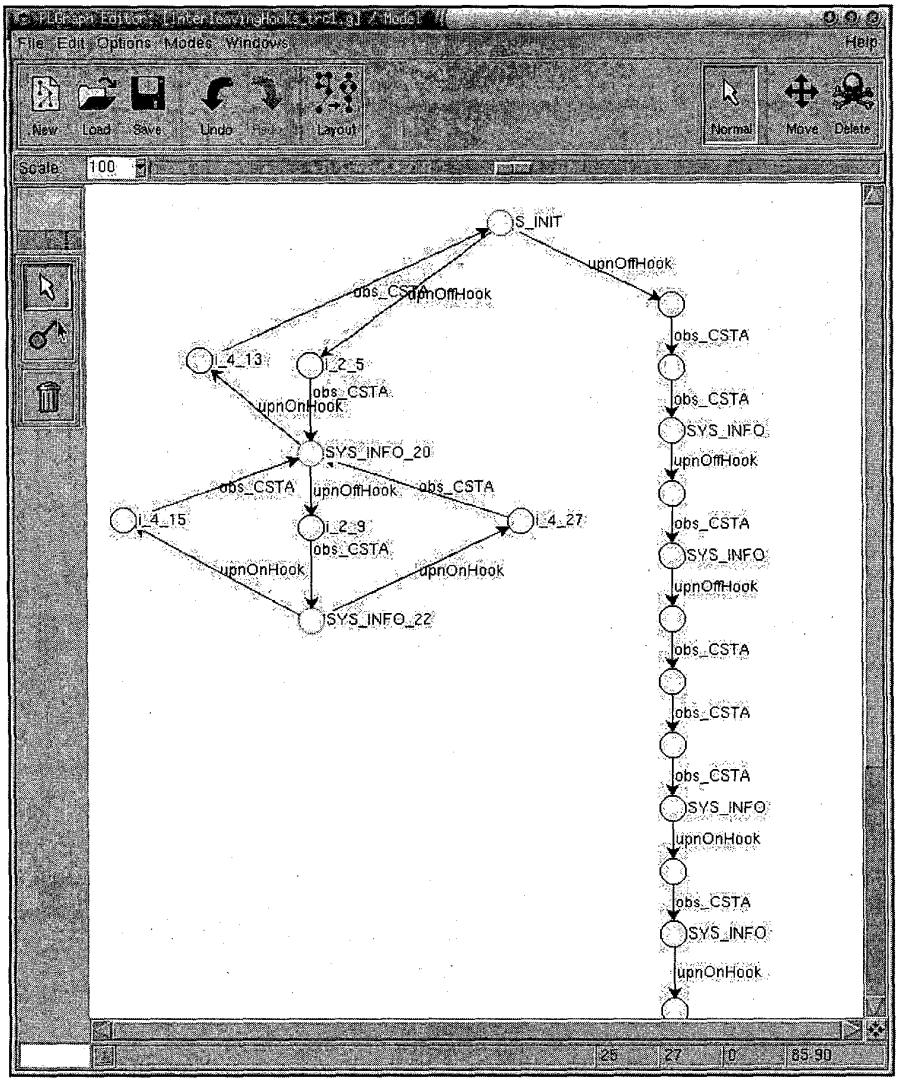

Figure 2: A Preliminary Model and a New Observation

There may be several different automata, each modelling some aspects only, which provide detailed views of some functions while disregarding others. We can view the collection of those automata as the system model: their combination into a single big automaton comprises the information gathered, but this automaton need not be computed explicitly. In the following, we assume that there is just one model of a system and ignore technical issues of its representation.

5.1 Regression Testing

Regression testing is a particularly fruitful application scenario for our approach. 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, our automatically constructed models, which in particular cover all test runs, can serve as

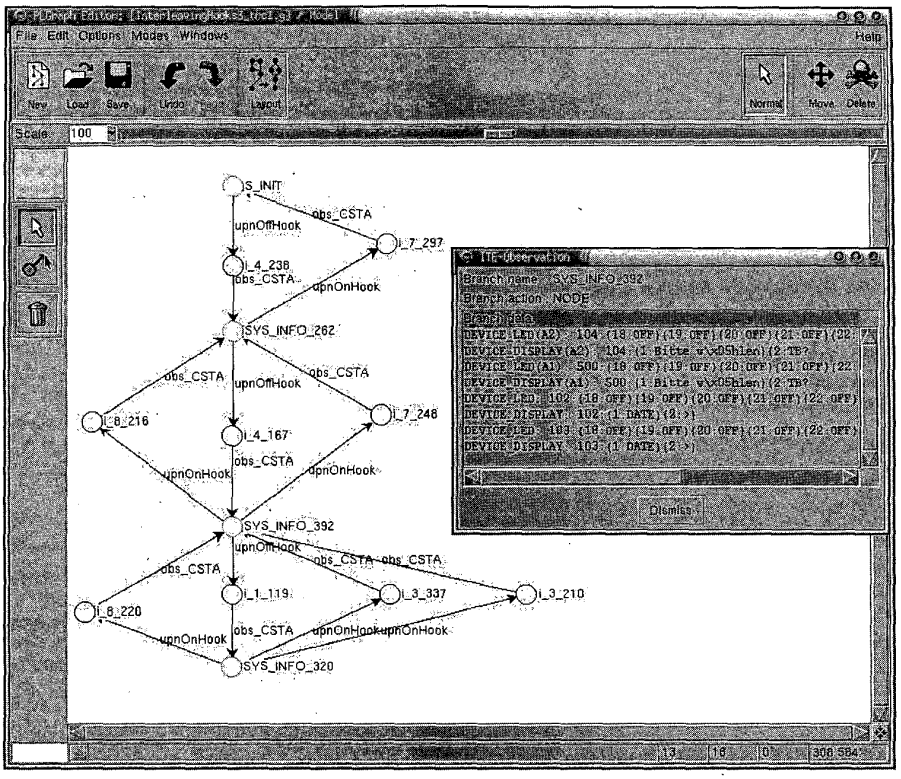

Figure 3: The Preliminary Model After Extension

a reference for the behavior of a new system version.

Note that abstract references like our models are much better than just protocols of old test runs: in old test runs, besides the few essential differences (mostly errors) there may be many inessential differences, and it is very hard to distinguish between these two types. Also, the completeness of models makes them more flexible than protocols: protocols become invalidated by changes to the test suite, whereas the model might still be applicable.

Some concrete benefits of using models in regression testing arise from the improved observation capabilities compared to manually constructed test suites, where test-case evaluation is mostly done via only few observation criteria and is therefore grossly incomplete.

The example in Fig. 4(left) shows a typical test case where two devices get connected. The evaluation of the test case is done by observation of the displays of the participating devices, which is sufficient for a correct system. However, an incorrect system might produce some erroneous signals on one of the other external interfaces, e.g. it is possible that one or more LED's are enabled while they should be disabled. In the model, *all* the visible information of the reference system in response to the test inputs are included, which provides us with valuable information during the test run/evaluation. Fig. 4 (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.

5.2 Test Coverage Analysis

The model-learning process starts with a number of test cases as observation seed, and continues to explore the system behavior automatically until a model consistent with all the observations is obtained. This automatic completion of the behavior extends the scope of the model beyond the range of the concrete test suite used. One may measure the portion of the system model covered by the existing test suite. This provides test engineers with an assessment criterion of the completeness of the test suite, and it also may

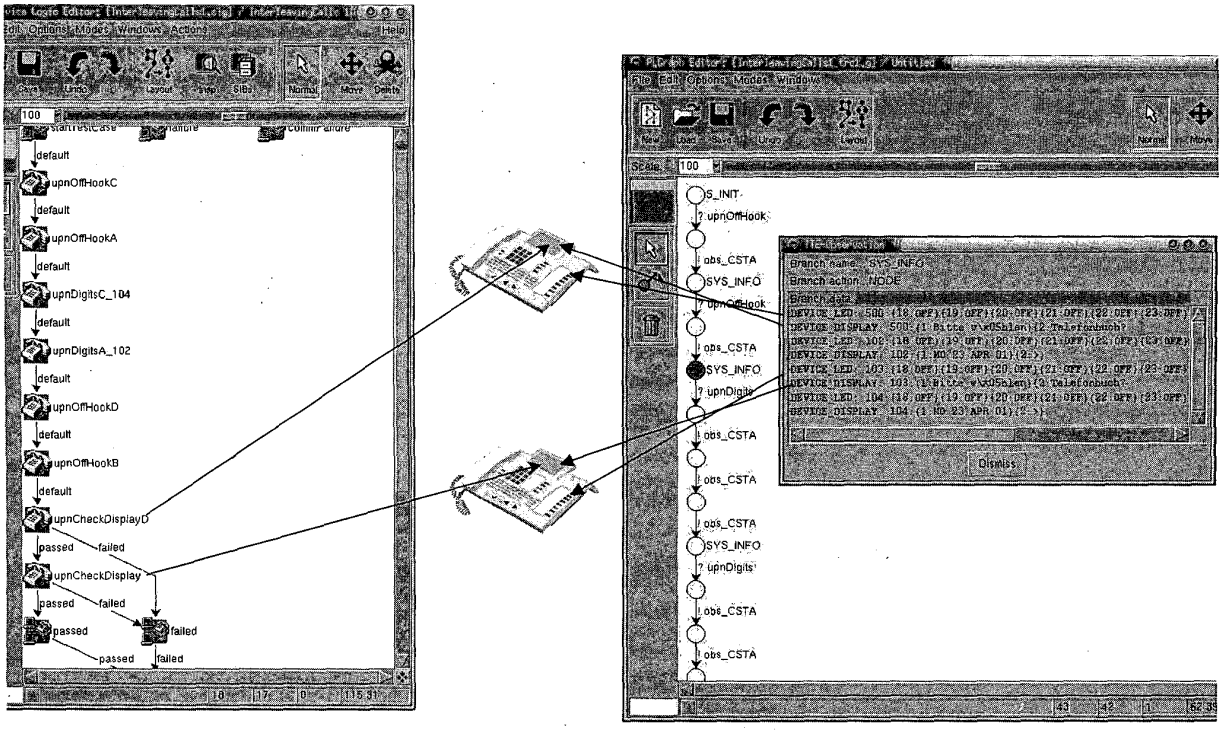

Figure 4: Enhanced Test Evaluation

provide information on how to extend it. In fact, automatic test generation techniques may be used here for automating this extension step.

5.3 On-line Test of Parameterized Systems

Besides the application to 'classical' testing, on-line test of running systems is particularly promising: a monitor based on an extrapolated model provides an ideal means to observe, and possibly remotely control and correct, a delivered system at the customer's site. The monitor lets the model 'run' in parallel with the system, tries to match the observed behavior in the model, and reports errors or warnings when the system behavior cannot be matched. The locations in the model where these warnings or errors appear typically provide diagnostic information, which can be profitably used for error correction.

In practice, a central problem is that models are constructed from observing small system instantiations, while a real system under observation has many more components, e.g. hundreds of phones instead of three.

Let k denote the number of phones represented in a given model. A simple but hardly satisfactory solution is to stop monitoring as soon as the number of components that have been activated so far exceeds k, and reenter monitoring once the system is reset. Indeed we can do much better: assuming a sufficient expressiveness of the model, which guarantees that the essential aspects of the system's behavior are covered (i.e. that the instance we learned is large enough to be representative), we can monitor as long as the observed number of *concurrently active* components is not larger than k. Online testing can continue even if more than k phones are active, if they can be partitioned into non-interacting *clusters* of size at most k. This requires a careful dynamic mapping of the concrete components of the system to components of the model.

5.3.1 Exploiting Symmetry.

This solution bases on the interchangeability of components, i.e. it exploits the fact that real systems are to a certain extent *symmetric*: often, a specific device can be replaced by another without behaviourally observable change. This symmetry consideration already enters our model construction technique [5]: we need not repeat observations for a particular phone line once

we have made it for a behaviourally equivalent one.

The online tester maintains for each cluster a mapping of active phones to symbolic phone identifiers in the model. It is thus able to follow their actions through transitions in the model and also to adjust the mappings. To capture formally the idea of interchangeable components, we use the notion of a *parametric* system $P(n)$, consisting of $n+1$ components $C_0, \ldots, C_n$, where C_0 is a central component and $C_1, \ldots, C_n$ are peripherals. Let us assume that messages between those components are records which, besides propositional information, reference components by their indices, and that runs of $P(n)$ consist of sequences of records. Then we say that the system $P(n)$ is *symmetric* if its set of runs is closed under arbitrary permutations of the $C_1, \ldots, C_n$.

This adequately captures abstract representations of systems like a switch C_0 with a number of phones connected to it, or an application server with an arbitrary number of clients, as in the considered CTI application domain. The theory underlying this kind of online testing is described in more detail in [16].

In this setting, based on a correct model, we can construct an on-line tester which checks the conformance of traces of a system under test with the modelled (reference) system. Discrepancies observed during online testing may in some cases be attributed to inaccuracies in the model and to the mapping procedure, but they are likely to give valuable hints on sources of problems in the installed system. Besides the model, the monitor device has two further constituents:

- a variable to hold the current state of the model,
- the above mentioned mapping assigning identifiers of peripheral components of the system being observed to the symbolic component identifiers (actor names) within the model.

If a model is indeed symmetric, checking whether some observed system trace is matched by one of the model can be done easily online:

- the identifier assignment is extended whenever needed and as long as free actor name are available (symmetry tells us that actor choice is arbitrary), and
- due to input determinism of the system and of its model, the next system state is unique (once the identifier assignment is fixed).

Online testing is computationally rather cheap, since matching does not require complicated calculations nor backtracking. So, we may even apply online test outside the test lab, observing deployed systems in the field to help in error diagnosis whenever something goes wrong. However, the scope of traces which can be matched is tied to the number of parametric components included in the model. In the test laboratory, only rather small system configurations are usually used. But for in-field test of deployed systems, the limits of the models may very soon be reached.

5.3.2 Strengthening Symmetry.

To further extend the range of matched behaviors we rely on closure properties stronger than pure symmetry, which can be indeed observed in many systems: between independent usages, a device typically returns to its initial state, at which it becomes again behaviorally indistinguishable from any other inactive device of that kind. This allows us to change the matching between inactive concrete components *dynamically, on demand*, and therefore increase the expressive power of a parametric model of size k: it captures all traces where no more than k phones are active *at the same time*. We call this more flexible style of matching *liberal*. To realize this flexibility, we must be able to decide at the model level when a device is inactive, resp. in its initial state.

Obviously, the set of liberally matchable traces can be much larger than the set of (strictly) matched traces. Therefore, on-line testers based on liberal matching will be able to follow the behavior of an observed system much longer. However, liberal matching requires more than symmetry of the system to yield correct results: if, for instance, the central component simply counts the number of different phones which have been active so far, and changes its behavior patterns once a certain threshold has been exceeded, a model built from observations on smaller-than-threshold instantiations can never incorporate this change.

This characteristic is inherent to our regular extrapolation approach, as well as to any analysis approach based on a bound observation depth (as in bounded model checking [2]): in practice, we may never arrive at a sound or complete model, if the diameter of the system is larger than the observations at hand and we never observe system behaviors that exceed the current model approximation. But even incomplete models of legacy systems prove to be extremely useful in practice, as they provide valuable information on real and potential bugs. In our experience, although some false alarms have been triggered in a testing scenario, the help by better error diagnosis and improved error detection far outweighed the nuisance of false negatives.

6 Results: Test-Based Model Learning in Practice

We have carried out model-construction experiments on several typical installations of the call-center application. For illustration purposes, we present four simple scenarios, each consisting of the telephone switch connected to a number of telephones (called "physical devices"). Figures 2 and 3 are snapshots from the learning process for similar scenarios. In each of these scenarios, the focus of the model was restricted to include a few actions of the telephones (inputs to the switch, formally denoted by A_I) and some responses of the switch (outputs, denoted by A_O). In the simplest scenario (S_1), there is just one phone permitted to lift ($\uparrow$) and hang up ($\downarrow$) the receiver; in the last (S_4), there are three phones where two (A and B) may establish a connection. For each of the first three, we also considered a variant scenario S_i' which includes an additional state-description output of the switch ($[hookswitch_D]$ for each device D).

S_1 1 physical device (A),
$A_I = \{A\uparrow, A\downarrow\}$,
$A_O = \{initiated_A, cleared_A, [hookswitch_A]\}$.

S_2 2 physical devices (A, B),
$A_I = \{A\uparrow, A\downarrow, B\uparrow, B\downarrow\}$,
$A_O = \{initiated_{\{A,B\}}, cleared_{\{A,B\}}, [hookswitch_{\{A,B\}}]\}$.

S_3 3 physical devices (A, B, C),
$A_I = \{A\uparrow, A\downarrow, B\uparrow, B\downarrow, C\uparrow, C\downarrow,\}$,
$A_O = \{initiated_{\{A,B,C\}}, cleared_{\{A,B,C\}}, [hookswitch_{\{A,B,C\}}]\}$.

S_4 3 physical devices (A, B, C),
$A_I = \{A\uparrow, A\downarrow, A\rightarrow B, B\uparrow, B\downarrow, C\uparrow, C\downarrow\}$,
$A_O = \{initiated_{\{A,B,C\}}, cleared_{\{A,B,C\}}, originated_A, established_B\}$.

Table 1 lists in the first two columns the number of states of the model resulting from the learning process and the number of membership queries L* would need to learn the model. Roughly, the number of membership queries is polynomial (between quadratic and cubic) in the number of states. The last two columns show the results of our optimized learning procedure, which exploits the specific profile of our application scenario, like prefix-closedness of the language and independence of certain actions. These optimizations, which lead to less than a quadratic number of queries per model state, are described in [?].

Particularly encouraging are the effort reduction factors of two orders of magnitude measured on the largest test scenarios (S_2' to S_4'), which are of typical size for the real life industrial test scenarios considered.

Table 1: Number of Membership Queries

Scenario	states	membersh. queries L*	optimized algorithm	reduct. factor
S_1	4	108	14	7,7
S_1'	8	672	30	22,4
S_2	12	2.431	97	25,1
S_2'	28	15.425	144	**107,1**
S_3	32	19.426	206	94,3
S_3'	80	132.340	288	**459,5**
S_4	78	132.300	1.606	81,1

7 Conclusion

We have presented a way of using test knowledge and test technologies to construct models of an unknown system under test that lacks a model. Our approach aims at empowering moderated extrapolation as a strong means to support the development, construction and maintenance of complex industrial systems. In particular, we have looked at the on-line test of a complex CTI system to show a pragmatic approach for dealing with parametric systems. Here it is important to find the right compromise in the trade-off between desired precision and cost of building a model. Validations systems that cover a large portion of the errors but with a high rate of false alarms (potential false negatives) are in practice often disabled, due to the high burden of dealing with false alarms. Well accepted are typically all low cost techniques that provide targeted error detection and issue warnings only when there are serious indications of potential errors. We are convinced that our approach which pragmatically employs formal methods like abstract interpretation, automata learning and model checking to semi-automatically construct formal models of running systems will change the industrial approach to system-level validation. In fact, we experienced that the ability to support the treatment of legacy systems turned out to be very convincing for the industrial adopters: There the need for a posteriori modelling techniques is often so strong that users are ready to take up new approaches and to contribute in refining them.

Acknowledgement We are very grateful to the ITE team, in particular to Oliver Niese, for discussions and fruitful comments.

References

[1] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 2(75):87–106, 1987.

[2] E.M. Clarke, A. Biere, R. Raimi Y. Zhu: *Bounded Model Checking Using Satisfiability Solving*, Formal Methods in System Design, Vol.19(1), pp. 7-34, 2001, Kluwer Ac. Publ.

[3] European Computer Manufactures Association (ECMA). Services for Computer Supported Telecommunications Applications (CSTA) Phase II. ECMA 217/218, 1994.

[4] European Computer Manufactures Association (ECMA). Services for Computer Supported Telecommunications Applications (CSTA) Phase III. ECMA 269/285, 1998.

[5] A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model Generation by Moderated Regular Extrapolation. *Proc. of the 5th Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2002)*, LNCS 2306, pp. 80-95.

[6] A. Hagerer, T. Margaria, O. Niese, B. Steffen, G. Brune, and H. Ide. Efficient regression testing of CTI-systems: Testing a complex call-center solution. In *Annual Review of Communication*, volume 55. Int. Engineering Consortium (IEC), 2001.

[7] Herakom GmbH, Germany, `http://www.herakom.de`.

[8] H. Hungar, T. Margaria, and B. Steffen Model Generation for Legacy Systems. Forthcoming.

[9] B. Lindner, T. Margaria, and B. Steffen. Ein personalisierter Internetdienst für wissenschaftliche Begutachtungsprozesse. In *GI-VOI-BITKOM-OCG-TeleTrusT Konferenz on "Elektronische Geschäftsprozesse"*(eBusiness Processes), Universität Klagenfurt, September 2001.

[10] T. Margaria, O. Niese, B. Steffen, A. Erochok. System Level Testing of Virtual Switch (Re-)Configuration over IP. In Proc. *IEEE European Test Workshop*. Corfu (GR), May 2002, IEEE Society Press.

[11] T. Margaria, O. Niese, B. Steffen. A Practical Approach for the Regression Testing of IP-based Applications. invited contribution to the volume *IP Applications and Services 2003: A Comprehensive Report*. IEC, Int. Engineering Consortium Chicago (USA), ISBN 1-931695-12-1.

[12] Toh Ne Win, Michael D. Ernst, Stephen J. Garland, Dilsun Kirli, and Nancy Lynch. Using simulated execution in verifying distributed algorithms. In *VMCAI'03, Fourth International Conference on Verification, Model Checking and Abstract Interpretation*. 2003, LNCS 2575, Springer Verlag pp. 283-297.

[13] Microsoft Corporation. Using TAPI 2.0 and Windows to Create the Next Generation of Computer-Telephony Integration. Whitepaper, `http://www.microsoft.com`.

[14] O. Niese, T. Margaria, A. Hagerer, B. Steffen, G. Brune, W. Goerigk, H.-D. Ide. An Automated Regression Testing of CTI Systems. In *Proc. IEEE European Test Workshop 2001*, pp. 51–57, Stockholm (Sweden), 2001.

[15] Rational, Inc.: *The Rational Suite description*, `http://www.rational.com/products`.

[16] B. Steffen and H. Hungar, Behavior-based model construction. In S. Mukhopadhyay and L. Zuck, editors, *Proc. 4th Int. Conf. on Verification, Model Checking and Abstract Interpretation*, LNCS 2575, Springer 2003.