To appear in Proc. 12th IEEE Intl. Workshop on Rapid System Prototyping (RSP'01), 6/2001

Model Based Testing in Evolutionary Software Development*

Alexander Pretschner, Heiko Lötzbeyer, Jan Philipps
Institut für Informatik, Technische Universität München, Germany
`www4.in.tum.de/~{pretschn,loetzbey,philipps}`

Abstract

The spiraling nature of evolutionary software development processes produces executable parts of the system at the end of each loop. We argue that these parts should consist not only of programming language code, but of executable graphical system models. *As a main benefit of the use of more abstract, yet formal, modeling languages, we present a method for model based test sequence generation for reactive systems on the grounds of Constraint Logic Programming and its implementation in the CASE tool AutoFocus.*

Keywords. *Cleanroom SW Engineering, Constraint Logic Programming, Extreme Programming, Incremental Development, Rapid Prototyping, Reactive Systems, Test Case Generation.*

1 Introduction

With the recent discussion on what has become known as Extreme Programming (XP, [1]), evolutionary development processes are gaining popularity. The main advantage of evolutionary processes is that each development cycle leads to executable code: This satisfies programmers who get instant feedback on their activities [4]; it also satisfies customers who can watch the results of the development process, and who can influence development from the beginning to the end.

XP is based on common —usually object-oriented— programming languages, which is in a way an advantage: XP is rather lightweight, as it makes little demands on the infrastructure (except for compiler turn-around time). Conversely, however, XP fails to leverage recent advances in CASE tools, graphical description techniques, or algorithmic approaches to verification or test case generation.

In this paper, we argue for an incremental development process for reactive systems based on higher-level description techniques. While reactive systems can, in principle, directly be implemented in programming languages such as C, we advocate the use of languages that are more abstract and closer to the application domain. We call such languages *modeling languages*, and artifacts written in this language (behavior) *models*.

In contrast to other work in this direction [3], we emphasize the use of a modeling languages with a clear semantics: Because of the importance of test cases as functional specifications, as debugging techniques and as a basis for regression tests after design increments, it is important to support test case development in all development phases. Thus, the main part of this paper (Section 4) describes a technique for the automatic derivation of test cases on the grounds of Constraint Logic Programming. Before, we give a general discussion of incremental design processes (Section 2) and present the modeling languages of the CASE tool AUTOFOCUS (Section 3) using a simple ATM example taken from [13]. In Section 4, test case generation is demonstrated with the same example, and the results are compared quantitatively to those of [13].

Related work is cited in the respective context. Different approaches to the generation of test cases are discussed in [9, 10, 13]; [13] also contains a discussion of testing as compared to (bounded) model checking.

2 Incremental Development

In this section, we discuss incremental software development processes and argue for the use of modeling languages instead of programming languages. We then discuss the role of testing in incremental development.

Development processes. One of the main difficulties in software engineering is that the requirements of the customer are prone to change while software is being developed. Spiraling, or evolutionary, development processes try to face this problem by building the software system *incrementally*, and by interacting with the customer after completion of each increment. Thus, requirements are not fixed at the beginning of the development, but instead converge during several incremental cycles and the respective feedback phases.

Common incremental process models include Boehm's spiral (meta) model [2], the Cleanroom Reference Model (CRM, [12]), Extreme Programming (XP, [1]), and rapid/evolutionary prototyping.

*Supported by the DLR (MOBASIS) and the DFG under reference numbers Be 1055/7-2 (KONDISK/IMMA) and Br 887/14-1 (INKREA).

The CRM relies on box models at each iteration of the development process. In each iteration, requirements are specified in a black box view that associates stimulus histories (inputs) with responses (outputs). In the state box view, these history/response relations are implemented by state machines; finally, the clear box view consists of actual code. All documents are subject to thorough reviews; requirements are traced through the documents. Software tests rely on statistical usage models (which, obviously, raises the difficulty of determining these models); in general, testing code in CRM is considered a means of measuring the success of the process rather than detecting errors in the system.

Extreme programming takes a less formal approach to incremental development. Apart from management issues or the pair programming principle, XP is based on two main concepts: Firstly, artifacts of the process are written as executable code in a common, usually object-oriented, programming language. Secondly, XP makes heavy use of testing. Programmers write test cases to ensure the plausibility of their code; they – preferably together with the customer – also write functional test cases to ensure that the code satisfies the customer's demands. Both classes of tests are supposed to be written before the code itself, and they are written in the same programming language.

An intuitive objection to incremental processes is that is is not always obvious in which direction development should proceed: As Michael Jackson puts it in the more general context of top-down development, the problem is that "you must already have solved the problem before the solution process is begun" [8]. Otherwise, the system will become cluttered, difficult to understand, and to communicate to the customers.

One solution might be to use standard reference architectures to guide development. In any case, the system description might need to be cleaned up after each increment; this approach is referred to as refactoring [5]. Refactoring is one of the foundations of XP where the absolute need for a-priori architectural design is denied and where the architecture evolves along with the product.

Neither CRM nor XP are formal in the mathematical sense. While the box model of CRM and its associated concept of refinement are said to be based on the mathematical notion of referential transparency, this is not reflected in the description techniques. The state machines of the state boxes have no formal semantics on which a refinement or equivalence concept could be based.

XP, on the other hand, is based on common programming languages, and explicitly disavows any connection to mathematical principles. Nevertheless, refactoring, which is essential in XP, requires at least an intuitive concept of equivalence of programs.

It is interesting that both CRM and XP make use of more than one kind of description of a system: Before executable code is written, CRM requires the specification of the system functionality as history/response pairs. In XP, system functionality is captured by functional tests. These functional tests can be regarded as specifications of the system to be designed while the plausibility tests are intended to check for programmers' errors such as forgotten cases, range errors, and possibly deficiencies, contradictions or omissions in the functional tests.

Model-based Development. We believe that modeling languages fit the demands of an incremental process better than programming languages: Their higher level of abstraction can lead to higher productivity of the programmers; their suggestive notations ease interaction with the customer and other programmers (even though code is sometimes better to understand than any other specification; the difficulty of understanding pure code is one of the reasons behind the XP principles of pair programming and collective code ownership). This is especially important for embedded systems, where hardware and software components have to be engineered simultaneously.

For embedded systems various forms of state transition diagrams, like Statecharts, SDL or the diagrams of ROOM or UML-RT are commonly used. We prefer the graphical specification formalisms of the CASE tool AUTOFOCUS [7], since they can be given a precise and —a point that is often neglected— a *simple* formal semantics. Precision and simplicity of the semantics are a prerequisite for the operational understanding of the models, for simulation and code generation, for the definition of an equivalence concept that can be the basis of refactoring patterns, for the integration with formal verification and validation tools [11], and —as we show later in the paper— for the automatic generation of test cases. Apart from a basic operational understanding, customers are not expected to be confronted with the formal semantics; we say that the behavioral models are *transparently formalized*.

With code generators, models can be translated to executable programs in languages such as C, Java, ADA or Prolog; thus, early lock-in to a certain programming language can be avoided. On the other hand, there is no reason to be dogmatic: Modeling languages can be integrated with handwritten code where convenient; models can serve as skeletons where custom code is filled in; sometimes, parts of the code for the final implementation will have to be rewritten for performance or memory space reasons. In these cases, however, the requirements will be clear, and this task seems to be easier than to use code from the beginning.

However, code generation is not the only application of models. With the simple semantics of AUTOFOCUS, it is possible to automatically extract test cases from the model by explicit specifications in the form of sequence or interaction diagrams; or implicitly from some structural coverage criterion. The first kind of tests corresponds to functional specifications or tests of XP, used for interaction with the customer; the second kind is useful for plausibility checks that help to discover inconsistencies or missing cases in the model.

Test cases from earlier design increments can be used as test cases for later ones, with a fully automatic assessment of whether or not parts of the system of the later increment conform to the earlier one. Moreover, if the system has to be recoded because the code generators turn out to produce inadequate code, the test cases can, in conjunction with handwritten ones, be used to test the final implementation rather than the model itself.

To summarize, we advocate a development process that consists of several iterations where at the end of each loop, instead of hand-written code an executable system model is presented to the customer. Coding by hand is deferred until the end of the development process, in case the generated code fails to satisfy performance or space requirements.

3 Behavior Models

AUTOFOCUS (`autofocus.in.tum.de`, [7]) is a tool for developing graphical specifications of embedded systems based on a simple, formally defined semantics. It supports different views on the system model: structure, behavior, interaction, and data type view. Each view concentrates on certain aspects of the specified model.

In AUTOFOCUS, a distributed system is a network of components that communicate via directed channels. Component networks are specified in *System Structure Diagrams* (SSDs). Figure 1 shows a typical SSD. Rectangles represent components, and arrows between them represent channels. Channels are typed, and they are connected to components via so called ports. SSDs can be hierarchically refined: Each component may itself contain a subnetwork. Ports of components in the top level network are meant to be connected to the outside world; they form the system's interface to its environment.

The *behavior* of a component is described by a *State Transition Diagram* (STD). Figure 2 shows typical STDs. Initial states are marked with a black dot. An STD consists of a set of *control states*, *transitions* and *local variables*. The local variables build up the component's *data state*.

A transition can have several annotations, separated by colons: a label, a precondition, input statements, output statements and a postcondition. The precondition is a boolean expression that refers to local variables and to pattern variables which are bound by one of the input statements. Input statements consist of a port name followed by a question mark and a pattern. Analogously output statements consist of a port name and an expression separated

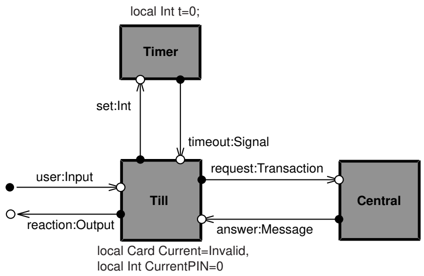

Figure 1. System structure

by an exclamation mark. Postconditions are assignments to local data state variables; they refer to local and pattern variables.

In addition to SSDs and STDs, AUTOFOCUS provides Message Sequence Charts (MSCs); they are used to describe the interaction between components, either for behavior specification (we are currently implementing an algorithm that converts them to STDs), for the visualization of simulations, and for the specification of test cases [13].

For the specification of user defined, possibly recursive, data types and functions AUTOFOCUS provides DTDs. The definitions in DTDs are written in a Gofer-like functional style (see Table 1).

Example. A simplified ATM system (Fig. 1) consists of three components: a timer, a central data base, and a till component (the actual ATM). Channel user serves as the ATM's input interface; a Card may be entered into the Slot, a function key FunKey with some associated Action may be pressed, or a PIN (an Int value) may be entered. Table 1 shows the associated data types. Users get the system's

Table 1. Data types/functions in the ATM

<table>
<tr><td>Card =</td><td>Invalid | Valid(getPIN:Int, Acc:Int)</td></tr>
<tr><td>Input =</td><td>Slot(Card) | FunKey(Action) | PIN(Int)</td></tr>
<tr><td>Output =</td><td>enterPIN | enterCard | enterAction
| timeoutError(Card) | byebye(money:Int,
Card) | ViewBal(Card) | errorWrongPIN</td></tr>
<tr><td colspan="2">Transaction, Message, Action, Signal = . . .</td></tr>
<tr><td colspan="2">fun message2output(Balance,C) = ViewBal(C)</td></tr>
<tr><td colspan="2">fun m2o(NoMoney,C), m2o(Money(M),C) = . . .</td></tr>
</table>

reaction via channel reaction (request for action, issuing money, displaying the balance). The timer component ensures that after a certain time the card is returned (e.g., in case something went wrong). Finally, the central database gets a request from the Till component on channel request and reacts accordingly (e.g., transmit balance, issue money, etc.). The complete system behavior is defined by the three component STDs in Fig. 2.

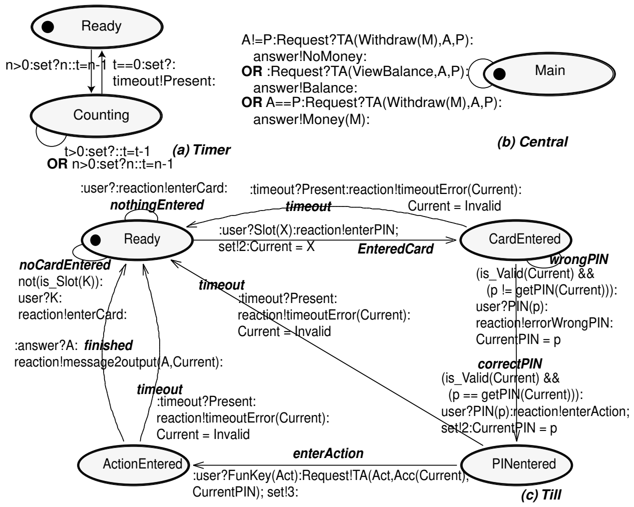

Figure 2. Component Behavior

Operational model. A transition can *fire* if its precondition holds and the patterns on the input statements match the values read from the component's input channels. After execution of the transition the values in the output statements are copied to the appropriate ports, and local variables are set according to the postcondition.

AUTOFOCUS components are timed by a common global clock, i.e., they all perform their computations simultaneously. Each clock cycle consists of two steps: First each component reads the values on its input ports and computes new values for local variables and output ports. Then, the new values are copied to the output ports where they can be accessed immediately via the input ports of connected components. This cyclic operation results in a *time-synchronous* communication scheme with buffer size 1.

4. Model Based Testing

Software testing is the process of executing a piece of software in order to detect incorrect behavior (a software failure). Software failures can either originate from wrong implementations or invalid or incomplete specifications. As testing requires some piece of software that can actually be executed, real tests are not performed until a set of basic modules have been implemented.

By using executable models in evolutionary prototyping, the testing process can start much earlier and can therefore be more effective. Model based testing covers both test case derivation from models as well as driving the test by executing the model. In the following we shortly introduce some test case generation techniques.

Test case generation. Test case generation starts with defining a *test purpose*. In general, a test purpose is an informal formulation of the properties which, later on, will be verified by the generated test cases. From *informal* test purposes, we derive a *formal* test case specification. Both test purposes and test case specification can be either functional (i.e., require test cases to test certain functionalities without regarding the model) or structural (e.g., require test cases to obtain a certain coverage of the implemented code). Then a number of actual test cases is computed for each test case specification. Simple examples for test cases include input histories, transition tours, traces, or constraints over them. We require test cases to be consistent with their test case specification, but they need not be executable [10].

To perform real tests, test cases must necessarily be executable and predict the results of the computation. Such test cases are called test sequences. A test sequence is an I/O trace that can directly be fed into an implementation or an executable system model, and after the test is performed, a *verdict* (pass, fail, inconclusive) can be assigned. Test sequences are rather operational whereas test cases are rather denotational, and it turns out to be nontrivial to determine actual sequences from arbitrary test cases which might be formulated as a set of constraints (in form of a message sequence chart, or a logical formula).

In addition to using counterexamples from model checkers that exhibit the disadvantage of no control over the generated trace, AUTOFOCUS offers two techniques for automatically generating test sequences from test case specifications: a propositional logic based approach and a CLP (Constraint Logic Programming) based approach. The propositional logic based test sequence generation compiles the AUTOFOCUS model and the query that restricts the system run into appropriate formulae which can efficiently be checked for satisfiability by a propositional solver such as SATO. The formulae include variables for each step of the model. In order to keep the formulae finite, the sequences are limited to a finite number of steps (cf. `ClockMax`, below). The desired test sequence is then generated by solving the formula. For details of the translation we refer to [13]. The problem of the propositional logic based approach is that the state space of the models has to be not only finite but also very small compared to actual problems. Although powerful abstraction techniques exist that allow for a significant reduction of the state space, it turns out to be very difficult to actually find a suitable abstraction.

The CLP based test sequence generation overcomes these limitations. Basically, the AUTOFOCUS model is translated into Prolog rules and constraints. The rules include symbols for input, output and local variables, and conditions are realized by constraints over the variables. By successively applying the rules, the model is symbolically executed, thus simulating one or more system run(s). By introducing additional constraints, the symbolic execution can be further restricted to get more specialized system runs. Note that unlike other approaches, we do not aim at extracting a test suite that is able to determine equivalence (or implication) between a specification and an implementation; *completeness* of test cases is thus not in our focus.

Translation into CLP. The (automatic) translation of AUTOFOCUS models into CLP code is straightforward. For each atomic component a step predicate is introduced. This predicate represents one single step of the component. Each transition of the corresponding automaton is represented by a single rule of the step predicate. This also includes the *idle transitions* – transitions that fire if no other can – which are not explicitly visible in the automaton. Basically the rules have the following form:

```
stepComponent(prevState, transition, Local-
Variables, Inputs, Outputs, nextState) :-
  precondition, postcondition.
```

where `prevState`, `transition`, and `nextState` are constants, `LocalVariables`, `Inputs`, and `Outputs` are variables or tuples of variables, and `precondition` and `postcondition` are predicates which may refer to all variables. Predicates are formulated as constraints.

For each composed component consisting of a network of communicating subcomponents, a special scheduler rule is created that drives the subcomponents and transfers messages between subcomponents and the environment. The scheduler rule has the same signature as the step predicates of the atomic components. Thus, from a black box view, no difference is made between an atomic and a composed component, and the encoding in Prolog rules reflects the component hierarchy of the AUTOFOCUS model. For the top level component, an additional rule is needed that successively calls the step rule and collects the histories of states, local variables, inputs, outputs, and transitions. The number of steps is limited by a variable `ClockMax` in order to avoid infinite runs. Details of the concrete translation are discussed in [9, 10]. In addition to the Prolog rules which model the transitions of the system, constraint solvers for the evaluation of the predicates (functions, data types) are needed. For integer type variables, constraint solvers for finite domain variables are available, and for the functional data types, a corresponding constraint solver is generated automatically by using Constraint Handling Rules (CHR, [6]). The advantages of the CLP encoding are obvious. Infinite data can be handled by using appropriate constraint solvers and a flexible search is performed by Prolog's backtracking and instantiation/labeling abilities. Beyond that, the translation preserves the structure of the model. We consider this an important point, as it is a prerequisite to adjust the search for special purposes (e.g. reach certain states or fire a specific transition, see below). In addition to a quite abstract encoding, the use of constraints entitles one to a-priori prune the search tree (e.g., testing properties that consist of implications), and to handle negative properties by delaying the respective instantiations.

In order to avoid loops in the process of execution, we added a simple mechanism: For each state, information about the transition that was last taken is retained; and when the state is revisited, another transition is to be taken (if possible). These sequences of transition orderings for each state turn out to be a powerful mechanism for goal-directed executions. One focus of our current work is more machine support for this objective (see below).

Experiments. All experiments in this paragraph have been performed on a SUN UltraSparc with 1GB memory, 400 MHz. In order to demonstrate the performance of our current system, we specify two simple test cases: a functional one and a structural one. For the functional test, we ask whether it is possible to withdraw any money at all - i.e., is there a run where the output channel eventually contains a byebye(M,C) message with $M \neq 0$. Regardless of the maximum length of the run, the system returns a run in $< .01$ seconds with a remaining constraint $M \neq 0$ and unspecified parameters of the inserted card. Since in this simple system no accounts are maintained, and one can thus withdraw any amount, the system can instantiate all values at will while satisfying the remaining constraint, e.g., $M = 1$. The same example fed into the SATO-based test sequence generator creates an instantiated test case in a time that is dependent on the maximum length of the run, e.g., in 54 seconds for a maximum length of 20 [13].

The structural test case specification consists of a run that contains all transitions in the Till component. The generated test cases exhibit a flaw in the model: Transition timeout from state ActionEntered to Ready should never fire for the central database immediately reacts once it received a request. However, if solely transition timeout is to be tested, the system discovers a run where timeout does indeed fire: The timer is set to 2 during transition to state PINEntered. Two ticks later, the user requests an action (ViewBalance). As a result, the transition to ActionEntered is taken, but at the same time, a timeout occurs, just before the timer is reset to 3. This causes timeout to fire. The problem is a simple error (a race condition) in the model, which can easily be corrected. Table 2 shows performance data (time and memory) for the transition tours with and without transition timeout for different maximum clock values. For comparison, the last two columns contain the system requirements for arbitrary runs of the given maximum length, c_{max} (which, with our technique of an alternating choice of transitions, yield in itself interesting test cases).

The SATO based tool cannot compute these sequences, but, *given the sequence of transitions*, it can find variable instantiations for the sequence. When the transition tour is given, SATO requires up to .55 seconds while the presented systems finds variable instantiations in $< .01$ sec. Note that the CLP based system outperforms this number for $c_{max} =$

Table 2. Statistics: Transition tour

	tour w/ timeout		tour w/o timeout		arb. c=c_{max}	
c_{max}	t[s]	m[KB]	t[s]	m[KB]	t[s]	m[KB]
40	1,908	497	.3	417	.01	232
60	.1	723	.1	638	.03	388
200	14.7	2,659	14.6	2,574	.3	2,030
300	1.0	4,631	1.0	4,546	.5	3,809
1000	22.5	4,250	22.8	4,094	5.4	4,804
9090	448	48,194	446	38,064	433	23,962
10^4	1,763	41,597	1,734	41,548	518	26,473

60 even though in addition to variable instantiations, it first has to determine the transition tour itself.

Obviously, performance in terms of time varies significantly with the choice of c_{max}. This is due to Prolog's depth first search strategy, where cutting the search tree at a depth of c_{max} obviously has an influence on how backtracking is to be performed. Our current work concentrates on determining this number; we expect the use of an A* search algorithm to yield satisfactory results.

The reason for the significant differences in efficiency between the SATO and the CLP based systems is most likely due to the fact that the translation into CLP is rather natural whereas the encoding in propositional logic is more complicated and imposes a noteworthy overhead.

5 Conclusion

We have presented our continuing efforts in specification based test sequence generation and its embedding in an incremental SW development process. The class of systems is neither restricted to finite nor to deterministic ones: recursive data types or real numbers are handled in exactly the same way as finite enumeration types (with the problem of finding appropriate instantiations); nondeterminism is handled by the backtracking mechanisms in CLP (with the problem of properly formulating verdicts). Experience with our industrial partners shows that customized management systems for (regression) testing are at least as important as a systematic generation of test cases; this is, however, not the focus of our current work. In order to assess the scalability of our CLP based approach, we are carrying out an industrial size case study with a large German manufacturer of smart cards; first results give us some reason to be optimistic but show that more intelligence in the process of search is necessary. As mentioned above, we are trying to adopt an A* search algorithm for this purpose. Furthermore, the intelligent instantiation of unbound variables in the computed test cases remains, w.r.t. a minimization of test cases, a difficult problem. We are trying to face it by means of simple heuristics, some of which are comparable to the interleaving strategy for the choice of transitions. Assistance in finding equivalence classes is mandatory for larger systems, in particular for mixed discrete-continuous systems. These issues form, together with ideas on compositional testing, the focus of current and future work in the area of test case generation.

One may well ask why we think coverage criteria on *models* are a good idea. Besides their potential use in debugging as well as regression testing activities, we hope to be able to use techniques similar to those for code generation in order to transform test suites on models to those on source code *while maintaining the respective coverage criterion*. This is a prerequisite for certification authorities to accept those test cases as conforming to the standard that has been applied in a particular project.

In terms of a sound notion of refactoring, we think that a formal semantics will help in building tools that support this process. This also requires a notion of refinement that is induced by the development process rather than by some mathematical theory (in fact, we see our work as a more formalized approach to the CRM). We consider a combination of a-posteriori validation steps (testing) and development steps that are sound by definition as a promising approach to systematically developing better, cheaper software.

References

[1] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 1999.

[2] B. Boehm. A spiral model of software development and enhancement. *Computer*, pages 61–72, May 1988.

[3] M. Boger, T. Baier, F. Wienberg, and W. Lamersdorf. Extreme modeling. In *Proc. Extreme Programming and Flexible Processes in SW Engineering (XP'00)*, 2000.

[4] F. Brooks. No Silver Bullet. In *Proc. 10th IFIP World Computing Conference*, pages 1069–1076, 1986.

[5] M. Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley, 1999.

[6] T. Frühwirth. Theory and practice of constraint handling rules. *J. Logic Prog.*, 37(1-3):95–138, October 1998.

[7] F. Huber, B. Schätz, A. Schmidt, and K. Spies. Autofocus—a tool for distributed systems specification. In *FTRTFT'96, LNCS 1135*, 1996.

[8] M. Jackson. *Software Requirements and Specifications*. Addison Wesley, 1995.

[9] H. Lötzbeyer and A. Pretschner. AutoFocus on Constraint Logic Programming. In *LPSE'00*, July 2000.

[10] H. Lötzbeyer and A. Pretschner. Testing Concurrent Reactive Systems with Constraint Logic Programming. In *Proc. 2nd workshop on Rule-Based Constraint Reasoning and Programming*, September 2000.

[11] J. Philipps and O. Slotosch. The quest for correct systems: Model checking of diagrams and datatypes. In *APSEC'99*, pages 449–458. IEEE Computer Society, 1999.

[12] S. Prowell, C. Trammell, R. Linger, and J. Poore. *Cleanroom Software Engineering*. Addison Wesley, 1999.

[13] G. Wimmel, H. Lötzbeyer, A. Pretschner, and O. Slotosch. Specification Based Test Sequence Generation with Propositional Logic. *J. Software Testing, Validation, and Reliability*, 10(4):229–248, 2000.