Proc. 2nd ICSE Intl. workshop on Automated Program Analysis, Testing and Verification (WAPATV), 5/2001, pp. 1–9

Model Based Testing with Constraint Logic Programming: First Results and Challenges*

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

Abstract

We summarize our continuing efforts at model based testing of reactive systems on the grounds of Constraint Logic Programming. First experimental results give rise to optimism w.r.t. scalability of our approach, point at necessary improvements, and they help identify future areas of research. Among others, these include search strategies more powerful than backtracking alone, appropriate (graphical) input languages for test cases, and theoretical aspects such as the relationship between test cases (traces) and system specifications.

Keywords. *Abstract interpretation, CASE, incremental SW development, reactive systems, refinement, automatic test case generation.*

1 Introduction

Even though testing is considered the most pragmatic and successful technique in (active) quality assurance, it still is a field that is insufficiently explored and systematized. One of the major issues in testing is a systematic and tool supported test case generation process that supports the development of high quality software systems.

Model based testing. In this paper we focus on test case generation for conformance testing. Traditionally, conformance testing aims at establishing evidence for the fact that an implementation conforms with the observable behavior of a specification (or a model in our case, i.e., an executable artifact written in a sufficiently abstract–e.g., graphical–language). In the context of an evolutionary software development process, conformance testing can also be used to establish conformance between models of different refinement levels as well as between different versions of the models (resp. implementations). Regression and conformance testing thus collapse. In addition, our approach also aims at finding test cases to locate errors in the model itself (as a debugging aid). We refer to all these testing activities as *model based testing.*

Contributions. The contribution of this paper is twofold. Firstly, we present the prototypical implementation of a tool for the automatic generation of test cases that is integrated into the CASE tool AutoFocus [18, 21, 22, 27, 25, 26, 20]. The underlying ideas–test case generation on the grounds of symbolic execution with constraint solving; separation of function definitions and control flow with finite state machines and test generation strategies for both aspects; not trying to prove equivalence or implication between two artifacts [30, 16] but rather focusing on incompleteness–are described and related to other generation algorithms.

Secondly, we discuss some design decisions our approach is based on at length. This leads to further questions such as the relationship with verification methods, formalized refinement/abstraction concepts, certification issues, and incremental development processes, thus pointing out our interest in the subject at the interface between theory and practical SW engineering.

Overview. In Section 2, we introduce the (modeling) concepts underlying the CASE tool AutoFocus. We then give a brief introduction to our approach to generating test cases on the grounds of Constraint Logic Programming [19]. Section 3 quite extensively discusses different aspects, application areas, advantages and drawbacks of the proposed approach. Issues that we consider important as being parts of future work are described in Section 4.

Related work. Related work is cited in the respective context. The work of Ciarlini and Frühwirth [7] as well as that of Marre and Arnould [23] are quite similar to ours. However, the further differs from our

*Supported by the DLR (project MOBASIS) and the DFG (project KONDISK/IMMA; ref.no. Be 1055/7-2).

work w.r.t. compositionality, the degree of automation as well as the combination of state machines with specifications in a functional language for guards and postconditions (the input language). The latter approach is based on Lustre as input language, and constraint solving is restricted to boolean and integer values. Both approaches do not rely on Constraint Handling Rules [15] for the definition of customized constraint handlers but rather predefined constraint handlers which, in our opinion, makes our approach more flexible. Recently, Meudec [24] has used Constraint Logic Programming (even Eclipse) for symbolic execution of SPARK (SPADE Ada Kernel) code in order to compute test cases. The focus is, however, neither on reactive nor on concurrent systems. In fact, SPARK is used for high integrity software and deliberately does not support tasks. Concurrency has then to be implemented explicitly by writing appropriate schedulers.

Other approaches to generating test cases are summarized in [34], and we omit their explicit discussion in this paper.

Our terminology is that of [22]. A *test case* represents a set of *test sequences*. Test sequences are I/O traces, and a test case can be a test sequence if it is fully instantiated. If it is not, constraints are necessary to define those sequences the test case describes.

2 Model Based Testing and AutoFocus

In this section we give a short overview over the AutoFocus description techniques and our approach to test case generation.

AutoFocus (`autofocus.in.tum.de`, [18]) 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, and is depicted by a certain kind of diagram.

The core items of AutoFocus specifications are components. A component is an independent computational unit that communicates with its environment via so called ports. The set of (typed) ports of a component describes its complete interface. Two or more components can be linked by connecting their ports with directed channels. Thus, component networks arise which are described by *System Structure Diagrams* (SSDs).

SSDs are hierarchic. This means that each component can again be described as a set of communicating subcomponents or another SSD, respectively. Atomic components are components which are not further refined. For these components a *behavior* must be defined which is achieved by means of a variant of Mealy machines (i.e., finite state machines with I/O).

Extended finite state machines are modeled by *State Transition Diagrams* (STDs). An STD consists of a set of *control states*, *transitions* and *local variables*. The local variables form the component's *data state*. Each transition has several annotations: a label, a precondition, input statements, output statements and a postcondition. Transitions can fire if the precondition holds and the input statements match the actual input vector at the component's interface (the input ports; remember that channels are typed and that these types are inductively defined in a functional language). Postconditions are assignments to local variables. After execution of the transition, the local variables are set accordingly, and the output port is bound to the values denoted in the output statements. Pre- and postconditions are defined with the help of a Gofer-like functional language that allows for the definition of possibly recursive data types and functions.

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.

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, for the visualization of simulations, and for the specification of test cases [34].

Test sequence generation. Specification based test sequence generation is the process of deriving suitable input/output sequences from specifications and usually informal test purposes. The determined sequences must be consistent with the specification and, later on, will be used to verify the properties specified by the test purposes on the implementation level. Test purposes may be either functional (i.e., test a certain functionality) or structural (i.e., obtain a certain coverage like branch coverage or path coverage). Due to the informal character of test purposes, it is impossible to compute executable test sequences directly from test purposes. Thus, test purposes have to be manually turned into formal test case specifications from which test cases can be derived. A test case specification can be formulated in many ways. Simple examples are input histories, transition sequences, traces, or constraints over them.

We chose to describe the test cases as constraints over traces which include all parameters of the system like input/output values, channels and variables, as well as reached states and fired transitions. This description technique is most general and is suitable for both functional and structural test cases. Test sequence generation is now defined as the task of finding suitable sequences which conform to the system specification and satisfy all constraints of the desired test case. We have automated this task by using Constraint Logic Programming (CLP).

Translation into CLP. The automatic translation of AutoFocus models into CLP (Eclipse, `www.icparc.ic.ac.uk/eclipse`) 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.

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 "doStep" 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 [21, 22].

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 as well as function definitions, a corresponding constraint solver is generated automatically by using Constraint Handling Rules (CHR, [15]).

Generating sequences. Test sequences are now generated by querying the Prolog system. An unconstrained query will result in successively computing all system traces, i.e. all traces that are consistent with the given AutoFocus specification. Each of these system traces can be seen as a test sequence which is possibly capable of finding errors in the implementation. The number of these test sequences is too large for an effective test strategy. Thus, the traces have to be constrained according to a given test case, and if possible, several traces should be executed simultaneously. This is done by transforming the test case into constraints over the appropriate history variables of the doStep-rule of the top level component (and these constraints can represent several traces, e.g., inequality constraints). Constraints can be defined and solved either by using predefined standard operators (e.g., boolean constraints) or by using more sophisticated self-defined constraint handlers (e.g., constraints that model temporal operators). Specialized constraint handlers can be effectively defined by using CHRs. Thus, the queries for computing actual test sequences consist of calling the doStep rule and further constraining its history variables. The Prolog system then symbolically executes the model. In this way more and more history variables get instantiated, and the appropriate constraint handlers are called. If the constraint handler fails to satisfy some constraints, the rule cannot be applied and the Prolog system performs a step backward, i.e., backtracks.

We kept the following three concepts mostly orthogonal: the basic executable model that reflects the specification, the additional constraints that define the test case and the search algorithm that guarantees an effective and efficient search strategy. They are orthogonal in the sense that altering one of them does not affect the others. The search strategy, for instance can be altered without changing the translation of the model. First experiences show that this design strategy allows for a very adaptive test sequence generation method.

3 Discussion

Our description of a system for the generation of test cases immediately raises some questions. We consider the most important ones to be those concerning scalability as well as those concerning a (quantitative) assessment of the system's performance in terms of efficiency and quality of the generated set of test cases. Space limitations necessarily render this discussion incomplete.

Scalability by interaction. A feasibility study [26] with a large German manufacturer of smart cards has given us some reason to be optimistic w.r.t. scalability of our approach. While there are principal problems with the explicit generation of the state space (cf. the discussion of the relationship with model checking below), this optimism is motivated by the *interaction opportunities* as provided by CLP, and, more particularly,

by CHRs (see also the treatment of appropriate input languages below). These interaction opportunities are founded on the fact that, with CLP/CHR, one can explicitly and simultaneously refer to the model *and* the property to be tested. With temporal logics, one cannot, for instance, explicitly formulate properties about the firing of certain transitions without instrumenting the underlying model.[1] Our approach, on the other hand, not only allows for taking into account observable behavior, i.e., inputs and outputs, but also for explicitly phrasing properties such as "do not enter state S" or "choose the transitions out of state S from a set T" *without altering the model*. This model can thus be interactively sliced (abstracted in an ad-hoc manner; we consider slicing in the sense of [29] for correct abstractions in the context of "programs", i.e., AUTOFOCUS models), resulting in an alleviation of the state space explosion problem. This alleviation is mainly based on CLP's facilities of a-priori pruning the search tree [19]. Our experience is that users are often capable of understanding the reason for the system's inability to compute a certain solution, and of using this information to "help" the system. We do not think that in the near future, push-button approaches to this kind of problem will yield satisfactory results; we consider interactivity as the key to a "graceful degradation" of our approach.

Performance and abstractions. When talking about performance, two issues arise: (1) Speed and memory requirements, and (2) quality of the generated traces. The first point is closely related to scalability. Experiments have shown that the choice of the maximum length of some traces is crucial w.r.t. the performance in terms of time ([25] contains an example where the time needed to compute a transition tour differs as much as four orders of magnitude where the maximum length differs by 20). This originates in Prolog's backtracking mechanism and renders the need for more intelligent search strategies obvious. Currently, we have implemented two different selection strategies. (1) In order to avoid loops that are caused by Prolog's depth-first search strategy, a simple heuristic was chosen: For each state the last transition that was taken is memorized, and when the same control state is re-entered, another transition is chosen (according to some user-definable pattern; the standard choice consists of trying one after the other, but sometimes it is useful to take a transition twice, etc.). (2) In our studies, it turned out that occasionally probabilities in choosing transitions are helpful (cf. [28]). Consider, for instance, a state with as many as fifteen emanating transitions and the test case requires one of them to be fired ten times in a row. A high probability for this particular transition helps in computing the corresponding test case.

As stated above, the opportunity to slice (or abstract) a model in an ad-hoc manner by prohibiting certain states or transitions is also crucial in an efficient computation of test sequences. We have implemented abstractions on the grounds of interval arithmetics similar to [17, 12] for integers and reals: Guards and postconditions are used in order to compute abstract data types (intervals) which, subsequently, yield induced operators on these abstracted data types (and the original operators in the postconditions may iteratively be applied in order to refine the abstraction and, consequently, the induced data types). This corresponds to abstract interpretation [9] on the level of function definitions; the power set lattice of the original data type and the–automatically computed–abstract data type can be shown to form a Galois Connection. Abstract interpretation on the level of transitions as, for instance, proposed by Dams et al. [11] is the subject of future work.

A problem with this approach is that computing induced functions on the basis of intervals only makes sense for data types with an intuitive notion of an interval. For inductively defined data types the induction order gives rise to "intervals" but seems less convincing than the ordering on numbers (is an abstraction for the "append" function on lists of length, say, zero and one, a good idea?). For embedded systems, however, sensor values usually are boolean or continuous, and the approach may well be applicable to this class of systems. Nonetheless, the usefulness of such abstractions still remains to be assessed.

In terms of quality of the computed test cases, a quantitative assessment is usually done on the basis of coverage metrics. A test purpose [22] can consist of the motivation behind such a metric (e.g., transition or state coverage), and the coverage criterion is used for the computation of test cases in an a-priori manner. Functional test purposes, as opposed to these structural tests, should origin in early phases of the development process; their "quality" is rather hard to assess quantitatively (they lead to a trace that exhibits the specified functionality, or they do not–"inconclusive" verdicts in our context occur for nondeterministic specifications). The use of coverage criteria for models is discussed below.

Incremental Development Process. With the advent of what has become known as Extreme Programming (XP, [3]), rapid/evolutionary prototyping ap-

[1] This is with the exception of Unity. Note that for other temporal logics, this choice is deliberate.

proaches have gained increasing popularity. The spiraling nature of such development processes exhibits, among other things, the advantage of early interaction with the customer and thus an early requirements validation: at the end of each loop, an executable (e.g., AutoFocus) model should be available. We consider the use of *modeling* rather than *programming* languages in a model based incremental development process the main prerequisite for a model-based generation of test cases. This is due to their higher degree of abstraction as well as the necessity of a formally defined semantics (cf. [25]; the ideas behind Extreme Modeling (XM, [5]) are similar to ours in terms of modeling, but our focus is rather on validation). The use of different specification formalisms is also found in the Cleanroom Reference Model [28]; our work in terms of development processes aims at a synthesis of the CRM, XP, XM, and classical prototyping approaches.

As a side benefit, an incremental development lends itself to model based regression testing: Test cases for an earlier increment can and should be used for testing later increments, and test cases for models have to be converted into test cases for implementations if code generators turn out to yield inefficient or inadequate code. However, the composition of components (one incremental step) may exclude some behaviors of one of the two components; computed test cases for the earlier increment will then lead to–erroneously detected–errors in the later one. This issue will be discussed later in the context of specified ("good") as opposed to unspecified ("bad") behaviors, or test cases, respectively.

Model Checking vs. Constraint Solving. We consider testing and model checking different approaches with different goals, advantages, and shortcomings [34]. Model checking based approaches are restricted to small, finite systems; they aim at mathematical completeness and soundness in terms of the significance of their results (cf. [6, 16, 30] for a relationship with different–formal–notions of testing as well as test case generation procedures that, unlike ours, aim at completeness). In an incremental development process, some increments may be so small that model checking can be applied to them. Model checking, however, usually does not exhibit a graceful degradation (see above). Instead, if a certain size of the model is exceeded, no results can be expected any more. The solution to this problem is abstraction (the "missing link" between model checking and constraint solving). Our test case generation technique exhibits some similarities with bounded model checking [4, 34] where the maximum length of the system runs in question has to be provided, too; on the other hand, it can, at least conceptually, cope with infinite systems (real numbers [27], recursive data structures [21]; see also [7, 30]). The a-priori-pruning technique as provided by constraint handling makes our approach somewhat similar to non-symbolic on-the-fly model checking, as, for instance, supported by the model checker SPIN. In [14], Fribourg points out the similarity between model checking, CLP, and abstract interpretation of infinite systems (e.g., hybrid systems [1, 32, 7, 27]; see also [10] for work on tabled resolution with XSB-Prolog and [33] for the relationship between top-down and bottom-up fixed point computations in LP). Model checkers are also used as generators of test cases. As stated above, we are not dogmatic about which technique is the better one. Instead, we see them as complementary approaches.

4 Future Work

The issues that have been discussed in the previous section give rise to a plethora of further questions, or future work, respectively. In order to structure these issues, we non-orthogonally divide them into theoretical, methodical, and implementation issues. These may overlap, as illustrated in Fig. 1.

Traces vs. Systems. We see the seemingly theoretical question of how to relate sets of traces with systems at the heart of our future scientific endeavors. It is motivated by the observation that the "equivalence classes" usually deployed in testing do rarely correspond to any sensible congruence relation–instead, they are usually extracted in an ad-hoc manner (cf. the selection criteria in [16]). This immediately led to the question for the relationship between a test suite and the system that is to be tested. Note that Tretmans, for instance, takes a conceptually different approach: starting with an implementation relation **ioco**, a test suite is generated that is able to decide whether or not an implementation belongs to the equivalence class of the system to be tested; and the notion of (observational) equivalence is defined by **ioco** [30, 6]).

However, in order to relate systems (automata) with I/O traces (test sequences or cases), traces have to be converted into corresponding systems or vice versa (cf. [8] for the relationship of different semantics). We do not consider the problem of relating finite and infinite traces (systems) here. When talking about reactive systems, the systems in question should obviously be input-enabled, i.e., be able to react to any stimulus in any situation. This leads to the notion of *completion*. In terms of mathematics, *chaos completion* en-

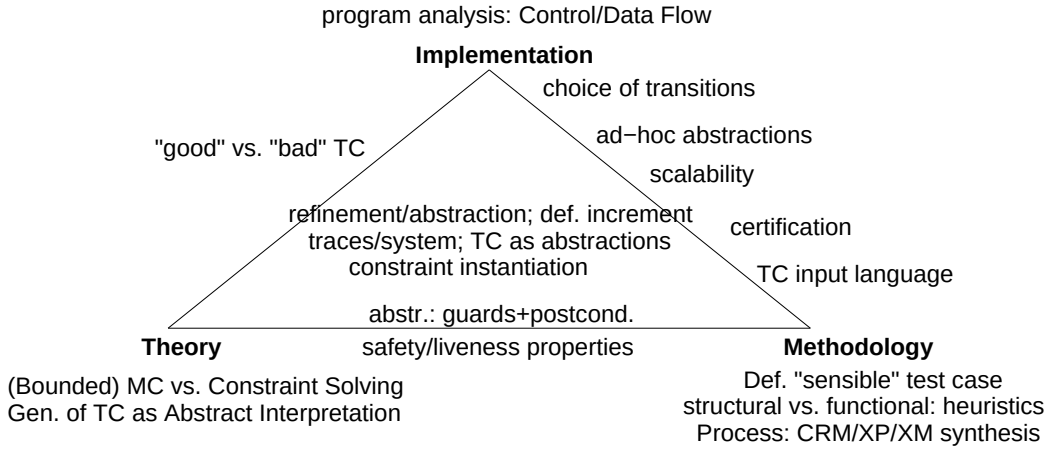

Figure 1. Issues in test case generation

ables one to define a natural concept of refinement on the grounds of trace inclusion or logical implication, respectively. Chaos completion allows the system to behave erratically (non-deterministically) whenever a non-specified event occurs. *Default completion*, on the other hand, makes the system remain in its current state whenever such a situation occurs: it idles (cf. the notion of repetitive quiescence in [30]). In terms of simulation environments (engineering), we consider default completion to be a better choice, however difficult the mathematical treatment of abstraction and/or refinement concepts may become.

The problem now is twofold. Firstly, for nondeterministic systems it becomes difficult to relate "concrete" with "abstract" systems. While chaos completed systems as induced by deterministic test cases are abstractions of the system to be tested, the corresponding trace inclusion relationship usually cannot be maintained for nondeterministic systems. In addition, for default completion such a simple formalization on the basis of trace inclusion is complicated. The idea here is to explicitly separate completed idle transitions from specified behaviors (δ outputs in [30]). Abstraction or refinement, respectively, may then be defined as trace inclusion modulo idle transitions; a transfer of the ideas of [30, 31] to the state machines with associated function definitions of AutoFocus may well solve these problems.

The refinement/abstraction relation induces a complete lattice of systems w.r.t. to the refinement ordering (the induced topology is at least a complete partial order for nondeterministic systems). The existence of least upper bounds in complete lattices may allow one to determine a semantic distance between different systems which, in turn, could lead to a new characterization of the quality of test cases. What we thus need is a concept of approximation for (extended) finite state machines.

This is particularly interesting for systems that are induced by "good" as well as "bad" test cases–test cases that correspond to system runs and test cases that do not. This can occur when systems are composed: The composed system may exhibit fewer or more behaviors than the single components alone. The question is then how to relate test cases for components with test cases for the composed system. Furthermore, when completed idle transitions are ignored, previously computed test cases may become impossible system traces. This problem is also related with the problem of incomplete specifications or implementations: forgotten else-branches, for instance, that are quietly "completed" with "default" behavior. However, problems seem to occur in parts of the specification that have been forgotten rather than in those that have been specified–test case generators should take this into account (e.g., by simple heuristics).

The above discussion also indicates that computed test cases may be used as abstractions of systems that can be used for further analysis.

Increments. The above discussion of a theoretical refinement relation (a topic that has been the subject of intense research in the last decades) raises the question of how this refinement relation is to be embedded in the understanding of an incremental development process. In particular, it has to be defined what exactly an increment (or an incremental step) is: Is it the addition of special cases to function definitions? Structural morphisms as propagated by Refactoring [13]? The addition of further functionality? Data refinement? Re-

finement in the sense of a top-down development as advocated by, among others, Dijkstra? Stepping from host to target architectures (A-B-C stages in prototyping)? A suitable formalization of some development steps that leads to increments that are correct by definition surely has to focus on some of these aspects and discard others.

Certification. The usability of formal test case computations is dependent on the expected payoff. We consider safety-critical systems one domain in which it actually may pay off. However, corresponding standards as, e.g., DO-178B for aircraft, require a set of test cases that is based on coverage on (object) code rather than on models (e.g., states or transitions). Test cases satisfying a certain coverage criterion on models have thus to be transformed into test cases satisfying a similar criterion on the code level (also cf. [31]). We believe that techniques similar to those employed in our code generators may be used for this purpose. However, the AutoFocus specific distinction between (Gofer-like) function definitions and functionality implemented by automata requires a thorough investigation of sensible coverage criteria even on the level of models. The main difficulty is that in AutoFocus, there is a separation between control and data states, where the latter can contain elements of the further. Transformations on the data space are usually achieved by referring to functional programs, and coverage criteria for a mixture of those programs and state machines remain to be defined.

Input language. One of the driving forces behind our approach is its practical usability (this is why we devote a lot of work to an efficient implementation). This also raises the question of suitable input languages for test cases. As indicated above, predicates on the basis of CHR surely are a good idea as a back end, but (graphical) notations (Message/Live Sequence Charts, automata, or even temporal formulae) have to be considered as input languages [34]. In terms of sequence diagrams and automata, for instance, constructs (and semantics) for negation will probably be needed. This also implies an investigation of the different properties (bounded liveness, safety) that are to be tested.

Applications. In addition to several toy examples (timer/blinker [21, 22], ATM [25], Mars Polar Lander [27]), we have used our approach for testing an inhouse smart card system in a feasibility study [26]. Furthermore, it has been used in the determination of test cases for firewalls [20]. However, experience shows that every example leads to new ideas on how to augment our test generation techniques. We are thus evaluating the approach with other medium-scale systems (currently, these include more smart card systems as well as a mixed discrete-continuous control system for the slats–leading edges–of a military aircraft wing).

Constraint instantiations. An issue that has not been addressed thus far is the intelligent instantiation of constraints. Our approach differs from that of Marre et al. [23] not only in the input language (Lustre vs. AutoFocus) but also in that we have implemented (recursive) function inversion and, thereby, explicit instantiations are performed *whenever* functions are evaluated. This leads to fewer constraints to be instantiated at the end of a symbolic execution ([23] explicitly uses boolean constraint solvers and thus enforces lazy evaluation of boolean connectives while we consider eager evaluation to be a more practical approach: "delaying" parts of logical formulae seems, to us, to occur at a too high level and, since not enough information is available, to significantly complicate the choice of suitable transitions). However, the problem remains of how to instantiate remaining constraints. Random testing has proven to be quite powerful in detecting errors, but since the set of test cases sometimes should be kept minimal, different heuristics have to be investigated (some of which exist). This issue is also related to that of certification and that of model based coverage vs. coverage on the level of implementations.

Search strategies. In the last chapter, the need for intelligent strategies for the selection of transitions was motivated. We consider the use of A*-like algorithms beneficial; the problem here is, however, the definition of suitable fitness functions. In terms of control flow on the grounds of control states, these may be based on the topological structure of the state machine, in terms of data flow, common techniques from program analysis seem good candidates for this kind of problem. Changing the communication semantics from a synchronous to an asynchronous one is likely to have an influence on the choice of transitions, too.

Implementation language. Our CLP based approach exhibits several advantages: relatively simple code generation, built-in backtracking, etc. In particular, the possibility of binding free variables is a prerequisite for our symbolic execution. However, more control over the search process would be desirable, and an a-priori lazy evaluation would be helpful, too (which is in particular the case for the translation of complicated function definitions as in the case of [20]). Functional logic languages such as

Curry (`www.informatik.uni-kiel.de/~curry/`) with Needed Narrowing [2] as its optimal operational semantics seem to be a better choice than Prolog alone. So far, the lack of efficient implementations as well as a connection with CHR or a comparable constraint solving meta tool has prevented us from discarding Prolog in favor of such languages. We conjecture that lazy functional languages will exhibit a poorer performance than Prolog (explicit representation of lists with data type extensions), but they surely facilitate the definition of different search strategies.

5 Conclusion

This paper is supposed to give an overview of the work in our group on testing and, more particularly, on test case generation. We have thus presented and extensively discussed an approach that is integrated in the CASE tool AUTOFOCUS. Experiments with other techniques than Constraint Logic Programming, namely propositional logics [34], have highlighted the need for interactivity in the process of generating test cases. Interactivity is naturally supported by approaches that are based on symbolic execution, and our first experiences shows that CLP and CHRs are well suited to this kind of generation technique.

Test case generation is just one problem in the testing process. We acknowledge that in practice, the current problems are concerned with test management issues rather than with test case generation. This is, however, not the focus of our current work.

Acknowledgments. We would like to thank Jan Philipps and Oscar Slotosch for fruitful discussions, and for commenting on a draft version of this paper.

References

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, February 1995.

[2] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symp. on Principles of Programming Languages (POPL'94)*, pages 268–279, 1994.

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

[4] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proc. TACAS/ETAPS'99*, LNAI 1249, pages 193–207, 1999.

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

[6] E. Brinksma. A theory for the derivation of tests. In *Proc. 8th Intl. Conf. on Protocol Specification, Testing, and Verification*, pages 63–74, 1988.

[7] A. Ciarlini and T. Frühwirth. Using Constraint Logic Programming for Software Validation. In *5th workshop on the German-Brazilian Bilateral Programme for Scientific and Technological Cooperation*, Königswinter, Germany, March 1999.

[8] P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Electronic Notes in Computer Science*, 6, 1997.

[9] P. Cousot and R. Cousot. Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM symp. on Principles of Programming Languages (POPL'77)*, pages 238–252, 1977.

[10] B. Cui, Y. Dong, X. Du, K. Narayan Kumar, C. Ramakrishnan, I. Ramakrishnan, A. Roychoudhury, S. Smolka, and D. Warren. Logic programming and model checking. In *Proc. PLILP/ALP*, Springer LNCS 1490, pages 1–20, 1998.

[11] D. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):22–43, 1997.

[12] S. Das, D. Dill, and S. Park. Experience with Predicate Abstraction. In *Proc. 11th Intl. Conf on Computer Aided Verification (CAV'99)*, 1999.

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

[14] L. Fribourg. Constraint logic programming applied to model checking. In *Proc. 9th Int. Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'99)*, LNCS 1817, Venice, 1999. Springer Verlag.

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

[16] M. Gaudel. Testing can be formal, too. In *Proc. Intl. Conf. on Theory and Practice of Software Development (TAPSOFT'95)*, LNCS 915, pages 82–96, Aarhus, Denmark, May 1995.

[17] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Proc. 9th Conf. on Computer-Aided Verification (CAV'97)*, 1997.

[18] F. Huber, B. Schätz, and G. Einert. Consistent Graphical Specification of Distributed Systems. In *Industrial Applications and Strengthened Foundations of Formal Methods (FME'97)*, LNCS 1313, pages 122–141. Springer Verlag, 1997.

[19] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. 14th. ACM Symposium on Principles of Programming Languages (POPL'87)*, pages 111–119, Munich, 1987.

[20] J. Jürjens and G. Wimmel. Specification-Based Testing of Firewalls, 2001. To appear in Proc. Andrei Ershov 4th Intl. Conf. on Perspectives of System Informatics.

[21] H. Lötzbeyer and A. Pretschner. AutoFocus on Constraint Logic Programming. In *Proc. (Constraint) Logic Programming and Software Engineering (LPSE'2000)*, London, July 2000.

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

[23] B. Marre and A. Arnould. Test Sequence Generation from Lustre Descriptions: GATEL. In *Proc. 15th IEEE Intl. Conf on Automated Software Engineering (ASE'00)*, Grenoble, 2000.

[24] C. Meudec. ATGen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution. In *Proc. 1st Intl. workshop on Automated Program Analysis, Testing, and Verification*, Limerick, 2000.

[25] A. Pretschner, H. Lötzbeyer, and J. Philipps. Model Based Testing in Evolutionary Software Development. In *Proc. 11th IEEE Intl. Workshop on Rapid System Prototyping (RSP'01)*, June 2001. To appear.

[26] A. Pretschner, O. Slotosch, H. Lötzbeyer, E. Aiglstorfer, and S. Kriebel. Model Based Testing for Real: The Inhouse Card Case Study, 2001. Submitted to FMICS'2001.

[27] A. Pretschner, O. Slotosch, and T. Stauner. Developing Correct Safety Critical, Hybrid, Embedded Systems. In *Proc. New Information Processing Techniques for Military Systems*, Istanbul, October 2000. NATO Research and Technology Organization.

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

[29] F. Tip. A survey of program slicing techniques. *J. Programming Languages*, 3(3):121–189, 1995.

[30] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software–Concepts and Tools*, 17(3):103–120, 1996.

[31] J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *7th. European Intl. Conf. on Software Testing, Analysis & Review (EuroSTAR'99)*, Barcelona, Spain, November 1999.

[32] L. Urbina. Analysis of Hybrid Systems in CLP(R). In *Proc. 2nd Intl. Conf. on Principles and Practice of Constraint Programming*, LNCS 1118, pages 451–467, Cambridge, Massachusetts, USA, 1996. Springer Verlag.

[33] D. Warren. Memoing for Logic Programs. *CACM*, 35(3):93–111, March 1992.

[34] 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.