Classical search strategies for test case generation with Constraint Logic Programming*

Alexander Pretschner
Institut für Informatik, Technische Universität München, Germany
www4.in.tum.de/~pretschn

Abstract

Test case generation for concurrent reactive systems on the grounds of symbolic execution basically amounts to searching their state space. As in the case of model checkers, different search strategies (depth-first, breadth-first, best-first, tabu) together with different strategies for storing visited states have a significant impact on the performance of the generation algorithm. We present experimental data for the performance of different search strategies and discuss the results, taking into account counter examples as generated by model checkers.
Keywords. CASE, Model Checking, State Representations, Symbolic Execution.

1 Introduction

Usually, errors in early software development design phases result in disproportionately high costs if they have to be corrected. This is due to the increasing number of implications of each design decision: the earlier a decision is made, the more implications it involves.

One commonly accepted solution to this problem is an incremental approach to software/ hardware development. The idea is to get possibly executable pieces of software (or models of hardware) in early phases that can be used for validation by interacting with a respective customer, and for verification w.r.t. given specifications (properties).

The focus of this paper is on reactive (embedded) systems, and we thus concentrate on behavior models. In order to get executable behavior models of such a system, a suitable high-level, preferably graphical, specification formalism is needed. Finite state machines have been identified as a practically usable candidate. With adequate CASE tools, systems can be built, simulated, and verified. Simulation not only helps the developer in understanding the system and detecting errors, but can also be used for customer interaction (validation). Verification and validation are usually achieved by testing, e.g., checking if, given a certain input, the system's output corresponds to the desired one.

Testing is, however, incomplete. Besides semi-automatic proof systems, model checkers aim at *guaranteeing* that the system satisfies a certain property. Basically, they exhaustively search the system's state space. This necessarily requires finite state spaces (e.g., SMV or SPIN) or built-in abstractions (e.g., Uppaal or HyTech for a restricted class of hybrid systems). The problem with state space explosion is obvious.

Along side these practical considerations, the concept of model checkers is such that they focus on verifying properties of models. Not only in the area of embedded systems the system model is just one step in the development process: the system has to be implemented after the specification (or model, respectively) has been validated. The implementation process may introduce errors that have to be detected. One instance of *conformance testing* amounts to showing that the behaviors of the implementation constitute a subset of the behaviors of the specification.

With this article, we continue a series of papers [21, 22, 26, 27, 28, 29] on model based test case generation for reactive systems specified with the CASE tool AutoFocus. The idea is to use the system model (a network of hierarchic components with associated behaviors) for the

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

generation of test cases that later on, are fed into the actual implementation (hardware). Note that test cases can also be used for validating the system *model* itself, but this induces the lack of an instance that decides whether or not the behavior as exhibited by the test case is a correct one–when testing the implementation, the model plays the role of this instance.

Our tool prototype takes an AutoFocus system description, translates it into Constraint Logic Programming (CLP) and symbolically executes the resulting system. This sometimes involves, however, a guessing procedure for the next transition to be taken. It is this guessing procedure that makes up the difference between search strategies such as breadth-first, depth-first, or best-first. In this paper, we re-examine an example from earlier work, apply different search strategies for the test case generation procedure, and compare them quantitatively.

Its contribution consists of a number of statistics that indicate the superiority of best-first search. If no suitable fitness function can be defined, *random depth-first search with global state storage* when deployed in a competitive parallel setting is a good choice. Furthermore, the paper discusses some conceptual differences between search strategies for model checkers with strategies for test case generation.

Related work. Incremental SW development processes include the Rapid Prototyping, the spiral (meta) model, Extreme Programming/Modeling [3, 4], and, more geared towards reactive systems, the Cleanroom Reference model (CRM, [30]). The benefit of models is particularly acknowledged in the Rational Unified Process [20] and the CRM. The embedding of model based testing in incremental processes is discussed in [27]. A theory of formal testing is tackled in [16, 5]. They share the commonality of defining an observational congruence ("selection hypotheses") on systems. Similar relations are used in [32, 31] to compute whether or not a system (model) conforms to its specification. We differ from this approach in that we do not want to prove such a conformance relation but rather approximate its proof as done in traditional testing. (Constraint) Logic Programming for test case generation has been used in [24, 7, 23]; our approach differs (1) in the class of systems we consider, (2) in the input language with a concept of interface and a combined approach to behavior specifications with automata and functional definitions on transitions, and (3) in the thereby induced necessity for powerful constraint handlers on the grounds of Constraint Handling Rules (CHR, [15]). Lutess [12] is a tool for the generation of test cases for Lustre (as is Gatel [23], see above). The difference with our approach is the use of model checkers or random number generators for the generation of test cases as well as a restriction to boolean data types.

Code generation on the grounds of CLP is, for various non-modular [25] automata considered in [17, 14]. The relationship of Model Checking and (C)LP with possibly tabled resolution procedures is discussed (and used) in [10, 13, 9]. Bounded Model Checking with propositional solvers for test case generation is considered in [33].

Test case generation on the grounds of mutation analysis is, among others, treated in [2]. In the context of mutation testing, constraints for the generation of test cases for transformational systems are used in [11]. The idea is to formulate constraints that approximate criteria for killing mutants.

[6] uses a mixture of BDDs and Presburger constraints for the representation of sets of states in reactive systems. [1] uses linear constraints on real numbers for model checking hybrid systems. Clearly, the focus is on model checking. The difference with our approach is that (1) we are mixing enumerative and symbolic techniques rather than computing fixed points on sets of constraints and (2), again, use CHR with constraint solvers on arbitrary domains (e.g., FD) for allowing convenient interactions and user-defined specifications of test cases.

Overview. The remainder of this paper is organized as follows. In the next section, we give a brief overview of the specification concepts of the CASE tool AutoFocus and introduce an example, taken from [28], that is used for assessing the different search strategies: a smart card for inhouse access control.

In the main part of this paper, in Section 3, we then present our approach to test case generation on the grounds of CLP. Several search strategies and state storage algorithms are discussed and assessed. We also examine the relationship between search strategies for test case

generation and for model checking. Section 4 concludes with an outlook on current and future work.

We assume familiarity with the basic concepts of CLP, finite state machines, and those of depth-first, breadth-first, and best-first search strategies.

Terminology. The terminology in this paper is that of [22]. A test sequence is a sequence of input/output event tuples. A test case describes a set of test sequences by imposing constraints on unbound variables in the I/O tuples. A test case specification is the formalization of a test purpose (e.g., "ensure coverage criterion C"); the aim of test case generation is to find test cases/sequences that satisfy a certain test case specification. Note that this may include the test of "unspecified" parts of the system's behavior.

2 AutoFocus

In this section, we briefly present the modeling concepts of the CASE tool AutoFocus (`http://autofocus.in.tum.de`, [18]) and present an example from earlier work that we will use as a basis for the generation of test cases with CLP.

AutoFocus. Similar to the UML-RT, systems are modularly specified by different views. The architectural view shows the system structure that consists of possibly hierarchic components interconnected by typed and directed channels. Typing is achieved by predefined or user-defined data types; possibly recursive types are defined by means of a Gofer-like functional language.

The bottom level component of each hierarchic component is equipped with an extended finite state machine, and each such machine can access the component's local variables. Transitions are equipped with a guard that accesses local variables as well as, by means of pattern matching, input channels of the respective component. The language of guards is the same functional language as mentioned above; it is thus possible not only to define functionality by means of state machines but also by means of possibly recursive functions. If the guard holds true, the transition may fire, resulting in an update of the component's local variables as well as the change of the current control state. Initial states are marked with a black dot; there is no such thing as an acceptance condition. If more than one transition can fire, one is selected non-deterministically; if none can, the system idles, i.e., remains in its current state.

Components are timed by a global clock, and they all perform their computations (firing transitions) simultaneously. Communication is synchronous (asynchronous communication is implemented by explicit buffer components).

In addition to the architecture, behavior, and data view, AutoFocus makes use of sequence charts. Even though we do not consider them further here, they play an important role in the development process: for specification, definition of test cases, and for the graphical representation of simulation results.

AutoFocus is equipped with code generators for Java, C, Prolog, and ADA. Furthermore, it is connected to a plethora of external tools. These include model checkers such as SMV or μcke as well as the propositional solver SATO, ATTOL/UnitTest for coverage measurements of test cases for generated ADA code, and DOORS for requirements tracing.

Example. Our example is the model of a smart card that may be used for inhouse access. After inserting the card and a personal identification number (PIN) into a terminal, the terminal may or may not, according to the rights a user owns, grant access to a particular door. This involves running authentication/identification protocols between card and terminal. A PIN may be blocked if the authentication process fails several times, and it can be unblocked if a corresponding personal unblocking key (PUK) is being entered. This example is part of an industrial case study we carried out in order to assess practical applicability of our tool prototype, and it is discussed in more detail in [28].

The card part of the system we consider consists of a single component that accepts commands on the input channel. These commands are provided by the terminal, and they include commands for authentication/identification, reading/writing data on the card, etc. The output is a signal

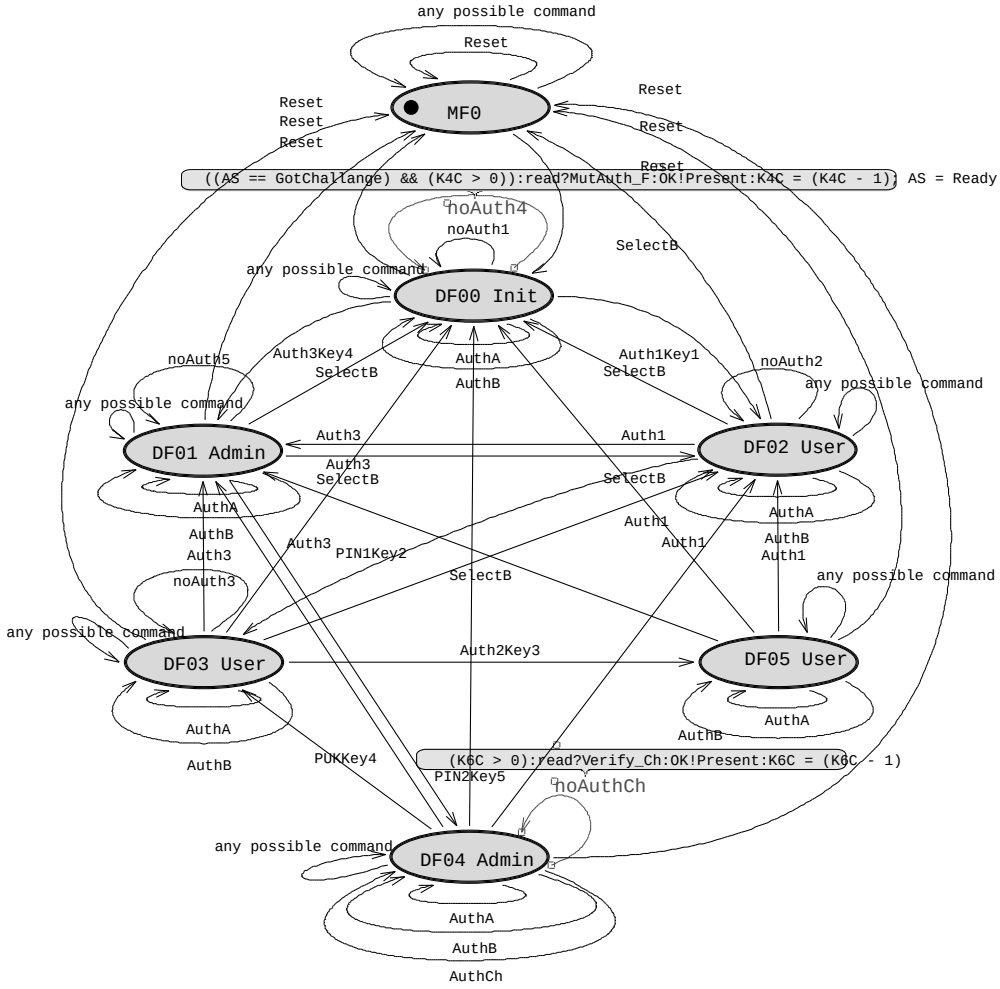

Figure 1: Inhouse Card [28]

that signifies whether or not the input command is a legal one. Its behavior is depicted in Fig. 1. Roughly speaking, the different states correspond to different access right levels to several parts of the card's data. The state is changed by an adequate authentication process. If this process fails, the respective counter (one for each state except for MF00) is decremented by one, and the card is locked if one of the counters reaches zero. By providing the corresponding PUK, it can be unlocked.

The test cases we will consider concern those that drive the card, for different counters, into the corresponding locked state. cnt_6 is the counter associated with state `DF04Admin` with initial value 15; cnt_4 is the counter associated with state `DF00Init` with initial value 14. The difficulty in finding a sequence that decrements cnt_4 to zero lies in the fact that between two decrements, a well-defined sequence of three transitions needs to be executed.

Note that this example contains a control state that is encoded by an internal variable. It stores the current status of the authentication process (three possible values). This reflects the experience that finite state machines alone quickly become too complicated.

3 Test case generation with CLP

In this section we first describe our algorithm for the computation of test cases based on symbolic execution with CLP. As mentioned before, this approach amounts to searching the system's state space. In our implementation, the part of the program that is concerned with implementing the search strategy is held rather orthogonal from its rest. We can thus modify the search

strategy without altering the rest of the code that is automatically generated from an AutoFocus specification. A user interface for specifying the search strategy to be employed is the subject of current work; thus far, a strategy with interleaved choice of transitions is generated automatically (see below).

After the description of the principal ideas, we discuss and assess several search strategies: (1) bounded depth first with (1a) naive left-right choice of transitions, (1b) interleaved choice, (1c) global and (1d) local state storage, and (1e) best-first choice of transitions. We also briefly discuss (2) breadth first search. In addition, we take into account the possibility of storing sets of states by means of constraints.

Code generation for CLP is presented in more detail in [21]; interleaving transitions in our setting has first been described in [27]. The use of constraints is described in [22]; the advantages of using constraints are more thoroughly discussed in [26].

(Constraint) Logic Programming. In Logic Programming, problems are specified as sets of first order predicates (disjunctions with at most one positive literal—implications). Common LP languages such as Prolog then interpret ("solve") these predicates in a procedural manner (resolution); backtracking mechanisms are built-in. In our context, the possibility of function inversion plays a crucial role: Under certain circumstances, given the result of a function application, one can infer the function's arguments (or a set of them). This is achieved by binding free variables and backtracking until these desired arguments are found.

However, there are some pitfalls in LP. On the one hand, solutions of programs (models of logical formulae) are always based on the same carrier set, a term universe (the so-called minimal Herbrand model). On the other hand, in implementations of LP languages, there is a certain order in which predicates are evaluated (in the procedural sense, see above) which may result in infinite evaluations even though the succeeding predicate could prevent infinite backtracking by imposing constraints that its preceding predicate can only satisfy in a finite number of ways. This led to the idea of merging Constraint Languages with LP into Constraint Logic Programming (CLP) languages [19]. These languages allow for the formulation of constraints that are checked for satisfiability in every step of the evaluation of a set of logical formulae (expansion of a node in the resolution tree), and they hence necessitate mechanisms for delaying subexpressions. This yields the possibility of a priori cutting the evaluation tree of these formulae; the "generate and test" paradigm of LP languages is modified to "constrain and generate". On the other hand, with CLP, one can calculate in domains other than the Herbrand universe, for instance finite (integer) domains $\mathcal{FD}$, or rational or real numbers $\mathcal{Q}$ and $\mathcal{R}$ (one crucial point in the latter two domains is to calculate on finitely representable intervals.) One can, for instance, formulate Linear Programming Problems with a set of unknown variables, and if the CLP language is equipped with suitable constraint solvers (e.g., Simplex), the desired optimal results can be found by binding variables to the corresponding rational numbers or intervals. LP is an instance of CLP with constraints being equations over terms, or finite trees, respectively.

Even though there are many constraint solvers available, it turned out that sometimes people do not want to calculate on one specific domain but rather a mixture of different domains, and that there sometimes is a need to create new domains and constraint handlers. This led to the development of Constraint Handling Rules (CHR [15]), a meta language that allows for the definition of new constraint handlers (solvers) that, subsequently, can be translated into the corresponding target language, CLP in our case.

Idea. Test case generation is achieved by running CLP code automatically generated from specifications. Since CLP allows for unbound variables as arguments, we run an ordinary simulation, but we do not specify inputs at all moments of time. In fact, by not restricting the system at all, we get an enumeration of all possible system traces.

Each bottom level component (components with associated behaviors) is translated into a set of predicates each of which models one particular transition. The predicates' heads include thus not only source and destination states as well as the name of the transition, but also formal parameters for histories of local variables, and for those input and output channels that are connected to the component.

Bottom level components (or rather the predicates that implement them) are run by a driver predicate that corresponds to the component that contains those bottom level components. Channels between components are modeled by local variables (existential quantification). This simple translation scheme can be applied recursively, and the driver predicates thus reflect the hierarchy of AutoFocus components.

In terms of function and data type definitions used for guards and postconditions, they are translated into constraint handlers or Prolog code and integrated into the model [21, 22].

As mentioned above, test case generation is done by simulating the system w.r.t. constraints as imposed by the test case specification. These constraints may refer to structural–such as "all transitions should be fired at least once" or "all control states are to be visited once"–or functional criteria (such as "find a trace that makes output o_1 happen and then output o_2" or "find a trace that includes input i and exhibits nothing but outputs from a set O"). These constraints are formulated by appropriate constraint handlers for Booleans or enumerative types (finite domains). In practice, this is, however, not enough for an adequate formalization of test purposes. User-defined constraint handlers based on Constraint Handling Rules [15] are a powerful tool for specifying test cases with predefined as well as user-defined constraint handlers, and they are integrated into our system.

While the possibility of a-priori-pruning the search tree is one advantage of using constraints, we consider the possibility of restricting the model to be the key to scalability: Often, designers know that for a particular functional test case, certain parts of the model are irrelevant; constraints allow to ad-hoc slice the model without actually altering it [26]. This also leaves room for techniques widely used in program analysis.

A further advantage of using constraints is that they allow for reducing the number of traces. As an example, consider a transition with a guard $i(t) = c_1 \vee i(t) = c_2 \vee \ldots \vee i(t) = c_n$ for input channel i at time t and commands c_j (elements of a data type "Command"). In a naive flattening Prolog translation of this fragment, n transitions need to be tried: i has to be bound to each c_j. Using constraints, we are happy with one single member constraint: $\{i(t) \in \bigcup_{j=1}^{n} c_j\}$.

In this way, a computed test case may represent a set of test sequences (those with $i(t) \in \bigcup_{j=1}^{n} c_j$ which might have been strengthened in the further computation). When testing the actual implementation, we need fully instantiated traces; how to perform this instantiation intelligently is, however, not the focus of this paper.

Naive $\rightarrow$. The simplest possible search strategy is a naive left-right-choice of transitions. As stated above, transitions are encoded by predicates that contain, among other things, the source and the destination state of a transition in their head. Choosing a transition can then be left to the evaluation mechanism of CLP: the transition predicate that occurs first is tried first, then the second, and so on. However, this approach is rather inefficient for it (1) revisits states that have been visited before and (2) is likely to run into cycles (see below).

Interleaving. In previous papers [27, 26], we have described our implementation based on a bounded depth-first search with an interleaved choice of transitions. This means that when a state is revisited, its outgoing transitions are not tried in exactly the same order as before; the list of transitions is rather shifted by one. If $Tr_{s,i} =< T_1, T_2, \ldots, T_n >$ is the sequence of transitions that are tried in that order from state s for the ith time, then $Tr_{s,i+1} =< T_2, T_3, \ldots, T_n, T_1 >$ is the respective sequence when s is visited the $i+1$st time. This simple mechanism, though imposing a little overhead, usually results in significant performance gains in terms of time since it decreases the likelihood of running into (control state) loops (Fig. 2 (a); with a naive left-to-right choice of transitions, the dashed ones are never taken.)

Storing states globally. One problem with naive bounded depth-first search is that states may be visited more than once. A state here refers to the cross product of tuples of data and control states for each component (one in the case of our example; the role of I/O as well as internal communication channels is discussed below). The obvious solution is to store each state once it has been visited. However, there is a price to pay: With this approach, the maximum

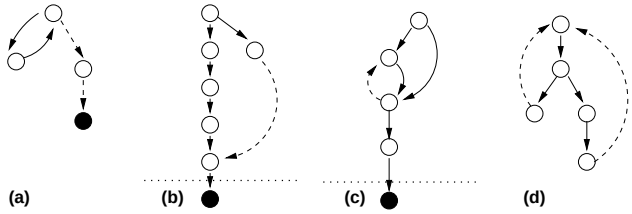

Figure 2: Search strategies

depth of the search tree becomes more important. In Fig. 2 (b), the black state that makes a trace satisfy the corresponding test case specification, is not reached (the horizontal line indicates the maximum depth). The reason is that the dashed transition is not taken for its destination state has already been visited and thus stored, indicating that it is not to be considered for further search. This is even though the black state could, on backtracking, be reached in fewer steps than the maximum depth. Note that for *testing* purposes, it is not even always desirable to exclude states from being visited twice; this reflects the strange kind of errors where something goes well n times, but not $n+1$.

Storing states locally. This motivates another search strategy. States are prevented from being revisited only when the depth at the moment of their storage is below the depth of the new visit (Fig. 2 (c) where the right transition from the initial state may be taken even though the corresponding destination state has already been visited). The problem with this approach is that it necessitates the storage of a large number of additional states; its benefit is that it usually allows for detecting shorter traces than the ones detected by strategies that store states globally. However, for our examples of decremented counters, no performance gains could be achieved. Since the number of possible traces increases with this local storage algorithm, the performance is rather significantly reduced.

Note that for structural test purposes such as "find a transition tour" the approach of storing states is not applicable. In Fig. 2 (d) the dashed transitions result in unsuccessful traces since the initial state has already been visited. A transition tour cannot be computed in this way. Also note that it depends on both the test case specification and the system whether or not the stored states need to include information about input and output channels.

Storing sets of states. The use of CLP enables one to store sets of states by simply storing the constraints that describe this set. Deciding whether or not a stored state *entails* a potentially reachable one is then crucial for deciding whether or not the respective transition needs to be taken. This issue is the subject of future work; [10] contains a discussion in the setting of deductive model checking with CLP.

To illustrate the idea, we consider a naive first implementation. Rather than storing each state $s = (ctl, auth\_stat, c_1, \ldots, c_6)$ (control state, data state for authentication process, six counters) separately, we look if there is already a stored state s' that differs from s in exactly one component. Say that the difference occurs in the control component, i.e., $s' = (ctl', auth\_stat, c_1, \ldots, c_6)$ with $ctl' \neq ctl$. We now delete s from the database and replace it by $(\{ctl, ctl'\}, auth\_stat, c_1, \ldots, c_6)$. If later on, we run into a state s'' which again, differs from s and s' in nothing but the first component, we delete the corresponding entry from the database and replace it by $(\{ctl, ctl', ctl''\}, auth\_stat, c_1, \ldots, c_6)$, and so on. Doing so for all components, this simple strategy results in a reduced need for memory: all components of the vectors except for the first just need to be stored just once.

Since storing sets of states is not the central issue of this paper, we do not go into further detail here. Two remarks are, however, in order. It is easy to see how the simple strategy can be made more powerful: Rather than restricting the search for already visited states to one component, we could look for the projection onto two or even more of them and update the database accordingly. This does require some further intelligence though: While different entries in the database correspond to a disjunction of states, the components of a state vector correspond to conjunctions: Consider a stored vector $(\{ctl_1, ctl_2, ctl_3\}, auth\_stat, c_1, \ldots, c_6)$ and a newly

$SP = sorted\_shortestPaths(s)$
forall $p \in SP$
  $T_{p[1]} = trans(p[1], p[2])$
  for $i = 1..\|SP\|$
    if $SP[i] \neq p$ then $T_{p[1]} = T_{p[1]} \circ trans(p[1], (SP[i])[1])$
return $\bigcup_{p \in SP} T_{p[1]}$

Figure 3: Transition ordering for best-first

visited state $(ctl_1, auth\_stat', c_1, \ldots, c_6)$ with $auth\_stat' \neq auth\_stat$. Clearly, we cannot simply coalesce both entries into $(\{ctl_1, ctl_2, ctl_3\}, \{auth\_stat, auth\_stat'\}, c_1, \ldots, c_6)$. We can, on the other hand, combine sets of states when it is sound to do so, but this requires efficient comparisons of sets of sets with the above mentioned suitable definition of entailment—obviously, we are looking for generalizations of Binary Decision Diagrams to domains other than the Booleans.

The second remark aims in the same direction. We will later see how an implementation of the above idea does indeed result in considerable reduction of allocated memory. This implementation stores states in the above mentioned manner but does not use them for the computation of traces itself. This would enable one to *compute with sets of states* rather than single states, something that is referred to as the "collecting semantics" in the literature on abstract interpretation (e.g., [8]). Knowledge of this computed collecting semantics may turn out to be a good starting point for the definition of an abstract semantics that can, in turn, be used for an actual abstract interpretation.

Best-first. As we will undermine quantitatively in the following, the ordering in which transitions from a given (control) state are taken is crucial for the performance of the test case generation. This issue becomes increasingly important when there are many transitions from that state. In some cases, it is possible to define a fitness function that w.r.t. a test case specification computes the (approximately) best transition to be taken. In the case of control states to be reached or transitions to be taken, this fitness function is comparatively easy to define: From each control state $c \in S$, we compute the shortest path p_c to reach the desired destination state s. We then implement a best-first search by defining the transition ordering for state c as follows. Transitions that connect c with the second state in p_c are tried first. For the remaining transitions emanating from c, we choose those that lead to a state d satisfying the requirement that there is no d' where the length of $p_{d'}$ is strictly shorter than that of p_d.

Iterating this procedure leads to a transition ordering that first tries transitions from c that are on p_c. It then tries those transitions that lead to states with minimum shortest paths to s, then those that lead to states with the second minimum paths, etc. This algorithm works because each subpath of an optimal path is itself optimal.

More formally, in Fig. 3, SP with length $\|SP\|$ is the sequence of shortest paths (sequences of control states) from all states $s' \in S$ to s. It is assumed to be sorted with the shortest path occurring first (this could be the path from s to itself; thus modeling the idle transition). Furthermore, it does not contain duplicates. For a sequence q, $q[i]$ denotes the ith element of q, and for sequences p, q, $p \circ q$ denotes their concatenation. $T_{p[i]}$ is the transition sequence that should be taken in this order when state $p[i]$ is to be left. $trans(s_1, s_2)$ returns a sequence of all transitions from s_1 to s_2 in arbitrary order. The existence of idle transitions prevents *trans* from returning the empty sequence in Fig. 3. Note that if there is no path from a state to s, it does not occur as the first element of a sequence in SP. This implies that transitions that lead to states that cannot reach s are (safely) ignored.

In our example, we are interested in optimal orderings w.r.t. reaching state DF00Init for decrementing cnt_4, and to reaching DF04Admin in order to decrement cnt_6. This is because transition noAuth4 is responsible for decrementing cnt_4, and transition noAuthCH is responsible for decrementing cnt_6. We thus use the algorithm for reaching states as one that is capable of finding good transition orderings for reaching transitions; as a further step, we move transitions noAuth4 and noAuthCH in front of the respective transition sequences.

Note that this simple heuristics is, in general, applicable only to control states for the number of data states may be too large. When the system contains data states, it is only a heuristic.

Table 1: $cnt_6 \rightarrow 0$

c_{max}		states	time	mem [MB]	len/succ
30	i	1982	14.8	3.0	30
	→	3348	36.1	3.5	–
	r	934-2424-4160	3.9-23.7-457.3	2.6-3.1-3.8	1/10
	bf	21	0.0	2.1	21
40	i	1442	8.3	2.8	40
	→	7137	157.6	5.0	–
	r	2149-5239-8399	18.0-91.3-209.5	3.0-4.2-5.4	4/10
	bf	21	0.0	2.1	21
50	i	1375	7.9	2.7	50
	n	10520	316.2	6.3	50
	r	1884-8183-14626	13.7-236.3-596.4	4.4-6.2-7.8	7/10
	bf	21	0.0	2.1	21
100	i	4789	65.7	4.1	100
	→	1346	7.7	2.7	100
	r	731-7860-15571	2.8-233-595.8	2.5-5.2-8.2	10/10
	bf	21	0.0	2.1	21
150	i	197	0.2	2.3	150
	→	1528	9.2	2.8	150
	r	446-7866-26714	1.2-285.8-1552.4	2.3-4.9-12.3	10/10
	bf	21	0.0	2.1	21
200	i	518	1.2	2.4	200
	→	1468	8.6	2.8	200
	r	3426-21279-62565	39.7-1542.6-6821	3.4-10.3-21.3	10/10
	bf	21	0.0	2.1	21
250	i	535	0.7	2.4	250
	→	2704	26.6	3.3	250
	r	359-13977-43397	0.4-744.6-3304.4	2.2-7.5-18.8	10/10
	bf	21	0.0	2.1	21
500	i	1518	4.6	2.8	281
	→	705	1.1	2.5	500
	r	5171-43030-115563	65.6-5289.2-20452.1	4.1-20.5-46.5	10/10
	bf	21	0.0	2.1	21
1000	i	1884	6.9	2.9	281
	→	900	0.9	2.6	739
	bf	21	0.0	2.1	21

This is the case in our example.

Breadth-first. The desire to get the shortest possible traces makes breadth-first search come into the game for it guarantees traces of minimum length. However, breadth first search severely suffers from memory explosion (at least if we do not employ symbolic representations such as BDDs). For the sake of a smaller memory allocation, we did not store the traces in the experiment. It is noteworthy that the architecture of our system does not necessitate the implementation of a naive meta interpreter (as usually done in breadth-first implementations in Prolog).

Experiments. We now give some experimental data, measured on a Pentium III with 850MHz and 256MB of memory. The CLP implementation we use is Eclipse (`www.icparc.ic.ac.uk/eclipse`) As mentioned above, we consider two functional test case specifications, namely decrement counters cnt_4 and cnt_6. Since a naive interleaving strategy without storing states as well as the strategy with local state storage resulted in prohibitive amounts of time needed, we omit mentioning the respective numbers here.

Concerning the test case that decrements cnt_6 to zero, Tab. 1 summarizes resource allocation for a strategy with globally storing states for interleaving (i), naive left-to-right (→), random

Table 2: $cnt_4 \rightarrow 0$

c_{max}		states	time [s]	mem [MB]	len/succ
300	i	986	2.0	2.6	300
	→	13138	420.2	7.3	300
	r	326-21488-56444	0.3-2146-7600.3	2.2-10.3-23.8	10/10
	bf	43	0.0	2.1	44
400	i	512	0.6	2.4	325
	→	34715	3284.6	15.4	400
	r	328-1319-6286	0.3-10.9-86.0	2.3-2.7-4.6	10/10
	bf	43	0.0	2.1	44
500	i	512	0.5	2.4	325
	→	22251	1386.4	10.7	500
	r	445-892-3397	0.5-3.5-27.4	2.4-2.6-3.5	10/10
	bf	43	0.0	2.1	44
1000	i	512	0.5	2.4	325
	→	1188	1.8	2.7	846
	r	383-604-1176	0.4-782-2120	2.4-2.4-2.7	10/10
	bf	43	0.0	2.1	44

(r) and best-first (bf) choice of transitions. For random choice, ten simulations have been run; the respective entries consist of min-mean-max triples. The first column shows the number of (globally) stored states, the second the time needed to find the first sequence satisfying the test case specification, the third the required memory (including 2.1MB required for the runtime system and the code), and the fourth the length of the respective trace (or, in the case of random choice, the number of runs that led to a successful trace). Since we consider bounded depth-first search in this case, we give data for several values of the maximum depth, c_{max}. Tab.2 contains the respective data for cnt_4.

Tab. 3 contains some data on breadth-first-search. Since the considered test cases lead to sequences longer than the maximum depth of 12, we just show the cumulative amounts of time and memory needed to proceed up to the specified depth.

In conjunction with breadth-first search, an implementation of the above mentioned simple approach to storing sets of states allows us, under the given resources, to enumerate states up to a depth of 14 rather than 12 (for a depth of 15, the virtual memory of 750MB gets again exhausted).

The last experiment we conducted consisted of manually slicing the model by prohibiting the system of firing those transitions that lead to decrements of the other four counters. This is done by means of constraints rather than by altering the model itself. The results are almost identical to those of best-first-search. However, the approach of imposing additional constraints is, in a sense, more general than using a best-first search: It may not always be possible to define a suitable fitness function. Slicing by constraints does, on the other hand, necessitate detailed knowledge of the system under consideration. We also used the approach of interactive slicing in the original study [28].

Discussion. As mentioned above, local state storage increases performance, but in our examples, the necessary resources still were too high as to consider this strategy a serious candidate. Its advantage lies is in potentially detecting shorter sequences.

Not surprisingly, breadth-first search results in an explosion of memory requirements. It is, however, a serious candidate for deductive model checking when storing sets of states by means of constraints together with approximation procedures as proposed in [10] are taken into account.

The experiments with depth-first search with global storage exhibit the commonality of finding rather long sequences (which is usual for depth-first search). As suspected, the choice of the ordering of transitions is crucial w.r.t. performance of the algorithms. This becomes evident with the results on random choice of transitions where the minimum and maximum performance in terms of time differs as much as four orders of magnitude. Storing states may lead to unsuccessful

Table 3: Enumerating states

depth	time [s]	mem [KB]
6	0.0	57
7	0.2	160
8	0.7	529
9	2.3	1,879
10	8.9	6,870
11	33.7	23,476
12	127.8	86,792
13	>600	>750,000

runs for a given maximum search depth even though in principle, the test case specification could be satisfied by a trace shorter than this maximum depth.

According great importance to the transition ordering is consistent with the data on best-first search. This strategy is able to compute short sequences with negligible overhead. In fact, it was able to compute traces of minimum length; regardless of of c_{max}, resource allocation remained constant. A simpler version of the fitness function that simply consists of always trying transitions noAuth4 or noAuthCH, respectively, first, and not caring about the ordering of the remaining transitions, leads to sequences that are a little longer, but can be found with comparable resources. It is likely, however, that this finding does not generalize when systems with more control states are to be tested. In fact, if a second version [28] of the model that replaces a data state by a set of additional control states (for the authentication protocols) is used for test case generation, the differences between the two fitness functions result in performance gains of one order of magnitude for the fitness function that is based on shortest paths. Furthermore, the automaton in this paper is heavily interconnected which explains why depth-first search with interleaving comparably efficiently produces traces for large maximum depths. In these cases, the larger the maximum depth, the more likely it is to quickly find a trace. In [27] we noticed that the choice of c_{max} has an important influence on the performance. Good heuristics for finding suitable maximum depths remain to be found.

An interleaving choice of transitions is usually preferable to an arbitrary fixed ordering; the advantage becomes negligible with growing c_{max}. This is due to the nature of depth-first search in highly interconnected graphs when augmented by a strategy that stores states.

With these findings, the numbers for random choice of transitions with a stunningly high standard deviation are easily explained. In the optimal cases (minimum requirements), the random choice is such that it is favorable for finding the particular test sequence.

These results suggests that if it is easy to define a fitness function, one should consider using a best-first strategy. If it is not, one should instead try a random choice of transitions (states globally stored) and perform several computations simultaneously. An additional process should use an interleaving choice. The first process to terminate successfully then kills the others. In our examples, that multiplies the minimum requirements by 11 (plus a little overhead for process scheduling), but this seems to be reasonable when taking into account the rather small requirements for the optimal solutions.

Finally, while our example is not a concurrent one, the encoding into CLP that keeps different components separate, is likely to assure that the results also hold in a concurrent setting. This claim remains to be verified; we are planning a new study with smart card systems that can best be modeled by concurrent components.

Model Checking. The resemblance of many issues discussed in this paper with model checking is eye-catching. This is for several reasons. Firstly, using the capability of model checkers to produce counterexamples for test case generation is an old idea. In fact, at least when they are reasonably short, they are used for localizing errors in specifications. Second, the problem of identifying a suitable search strategy does constitute large parts of work done in areas such as non-symbolic on-the-fly model checkers like SPIN. Thirdly, and maybe less obviously, the main

problem consists of computing (approximations of) fixed points or subsets thereof. We will briefly address these issues in the remainder of this section.

Model checkers, at least without built-in abstraction capabilities as found in Uppaal or HyTech, are inherently restricted to small finite systems. A CLP based approach to test case generation like ours is not. This is because we explicitly construct parts of the state space on-the-fly rather than first constructing and then computing fixed points on it as done in symbolic checkers. Constraints–in a sense, a generalization of boolean formulae encoding transition relations and encoded by BDDs–are useful in at least two ways. Firstly, they allow for storing possibly infinite sets of states (see also [10, 13], the latter of which notes that the boundary between traditional model checkers and CLP systems has become blurry, as in the case of HyTech). Secondly, powerful self-defined constraint handlers like CHRs can be used for interactively slicing the model [26]. Again, we consider this a key to scalability of our approach.

The problem of finding suitable search strategies is very similar in non-symbolic model checking. In fact, as bounded depth-first search has become an option in available tools, our approach is not really different. At least conceptually, there is a slight difference between model checking and generating test cases for the latter *always* tries to find a witness. Model checkers usually aim at proving a property of an entire system. This difference allows us to adopt the search strategy w.r.t. a given test case specification (note that this is also principally possible for on-the-fly model checkers, but this requires exact knowledge of the respective search algorithm). It is noteworthy that in testing, one is mainly interested in existentially path-quantified properties. The approach of first generating the entire state space may thus not be the prime solution in this context.

Again, the use of constraints can help in storing possibly infinite sets of states which, in conjunction with a suitable concept of entailment, may lead to more powerful search strategies. This claim remains to be verified empirically. Note that partial reduction, as used in SPIN, is unlikely to yield performance gains for AutoFocus systems are inherently synchronous. However, we consider it interesting to apply partial reduction to test case generation for discretized mixed discrete-continuous (or hybrid) systems.

Finally, CLP with suitable memoing techniques can be used directly for model checking [9, 10]. Results from the area of deductive databases (e.g., bottom-up evaluation with magic templates) may prove powerful also for model checking reactive systems. The idea is to generate fixed points or approximations thereof in a bottom-up manner. This results in deductive model checking procedures for possibly infinite systems (by means of widening/narrowing operators). While thus far we have concentrated on test case generation, our existing infrastructure directly provides an experimentation platform for deductive model checking (e.g., with the post-μ calculus and appropriate query logics).

To summarize, model checking and test case generation (on the grounds of CLP) are different w.r.t. their purpose, and it seems reasonable to advocate a complementary use of them. While they are quite similar in terms of problems related to search strategies, the use of constraints may prove a powerful tool to handle complex systems.

4 Conclusion

We have presented experimental data for several search strategies implemented in our CLP-based tool prototype for generating test cases from AutoFocus system specifications. An example from an earlier feasibility study was used to identify circumstances under which certain strategies are likely to perform better than others. Furthermore, we have discussed the relationship of model checking with our approach that is based on symbolic execution with constraints. The main differences are grounded on the handling of infinite (or large) state spaces as well as the opportunity of using constraints to naturally slice models (without actually having to alter them).

There is a plethora of future work to be done in this area. The need for research in the area of good fitness functions for different system topologies and/or properties to be tested is obvious. In addition, we consider efficient strategies for storing sets of states as a prerequisite for handling very large systems, too. In the following, we pick some additional aspects from a recent position paper [26].

One issue is the use of our approach to generating test cases for mixed discrete-continuous

systems [29]. Storing sets of states as well as partial order reduction techniques seem promising candidates to successfully tackle this particularly difficult class of systems.

The exact relationship between a set of test cases and the system to be tested is not entirely clear. Rather than defining a congruence on traces and use this congruence to generate tests, we generate tests and are interested in how they relate to the system (or specification) to be tested. This is likely to result in the definition of a suitable approximation order on finite state machines (classical notions of refinement/abstraction with chaos completion pose problems when lifting traces to systems and being concerned with resulting input-enabled nondeterministic systems; the role of idling–or δ [32]–transitions makes such systems difficult to embed in a refinement context).

In addition, suitable input languages for test case specifications are needed. One might consider message or live sequence charts, but they would require an intuitive notation for and semantics of negation.

The problem of finding good heuristics for instantiating test cases into test sequences has been mentioned above. For numeric values, one can use limit analyses, e.g., instantiating an interval to three single values.

This example also leads to a notion of "bad test cases". Experience shows that specifications are incomplete (while a simulation semantics, in particular with idle transitions, suggests completeness), and that errors tend to occur where designers forgot to specify some special cases (e.g., a missing else-branch in an if-statement). Test case generators should be able to compute test cases that test these cases; this is related to (a) the role of idle transitions and (b) the definition of suitable coverage metrics. For state machines with functional definitions as allowed in AutoFocus, such metrics do not, to the author's knowledge, even exist.

Furthermore, generating large sets of test cases to satisfy some coverage metrics only makes sense if the generated test cases on the level of models can be lifted to the implementation level (i.e., generated code, or hand-written code when generators produce inadequate code) *by maintaining the respective coverage criterion*. On the implementation level, such test cases are used by certification authorities to verify conformance with a given standard (e.g., DO-178B for aircraft).

With a component-oriented specification language, we consider a compositional approach to test case generation promising: Generating test cases for (simple) components and combining them into test cases for larger systems. The problem is that a test case for one component may (and often will) be a forbidden behavior of the composed system.

Finally, a model-based development process and automatic test case generation needs to pay off. Some of our industrial partners in the area of safety-critical systems seriously consider implementing such a process which will provide an opportunity to assess benefits and problems.

Acknowledgment. The tool prototype is developed together with Heiko Lötzbeyer whose contributions are gratefully acknowledged. Oscar Slotosch built the original model [28].

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] P. Ammann and P. Black. Test Generation and Recognition with Formal Methods. In *Proc. 1st Intl. workshop on Automated Program Analysis, Testing, and Verification (WAPATV'00)*, pages 64–67, 2000.

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

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

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

[6] T. Bultan. *Automated symbolic analysis of reactive systems*. PhD thesis, University of Maryland, 1998.

[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 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.

[9] 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.

[10] G. Delzanno and A. Podelski. Model Checking in CLP. In *Proc. Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, pages 223–239, 1999.

[11] R. DeMillo and A. Offutt. Constraint-Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.

[12] L. du Bousquet and N. Zuanon. An overview of lutess, a specification-based tool for testing synchronous software. In *Proc. 14th IEEE Intl. Conf. on Automated SW Engineering*, October 1999.

[13] 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.

[14] L. Fribourg and M. Veloso-Peixoto. Automates Concurrents à Contraintes. *Technique et Science Informatiques*, 13(6):837–866, 1994.

[15] T. Frühwirth. Constraint Handling Rules. In *Constraint Programming: Basics and Trends (LNCS 910)*, pages 90–107. Springer Verlag, 1995.

[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] G. Gupta and E. Pontelli. A Constraint-based Approach to Specification and Verification of Real-time Systems. In *Proc. IEEE Real-time Symposium*, pages 230–239, San Francisco, December 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 M. Maher. Constraint Logic Programming: A Survey. *J. Logic Programming*, 20:503–581, 1994.

[20] P. Kruchten. *The Rational Unified Process - An Introduction*. Addison Wesley, 2000.

[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] O. Müller and T. Stauner. Modelling and verification using Linear Hybrid Automata. *Mathematical Computer Modeling of Dynamical Systems*, 6(1), March 2000.

[26] A. Pretschner and H. Lötzbeyer. Model Based Testing with Constraint Logic Programming: First Results and Challenges. In *2nd ICSE Intl. Workshop on Automated Program Analysis, Testing, and Verification (WAPATV'01)*, May 2001. To appear.

[27] 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.

[28] 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.

[29] 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.

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

[31] V. Rusu, L. du Bousquet, and T. Jéron. An Approach to Symbolic Test Generation. In *Proc. Integrated Formal Methods*, 2000.

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

[33] 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.