Design Constraints in Symbolic Model Checking

Matt Kaufmann*, Andrew Martin, and Carl Pixley

Motorola, Inc.
P.O. Box 6000, MD: F52
Austin, TX 78762
carl_pixley@email.mot.com

Abstract. A time-consuming and error-prone activity in symbolic model-checking is the construction of environments. We present a technique for modeling environmental constraints that avoids the need for explicit construction of environments. Moreover, our approach supports an assume/guarantee style of reasoning that also supports simulation monitors. We give examples of the use of constraints in PowerPC$^{\mathrm{TM}^1}$ verification.

1 Introduction

This work addresses the problem of providing a convenient way for designers to specify environments for symbolic model-checking, while supporting assume/guarantee reasoning.

CTL model-checking [1] is defined with respect to a closed system, that is, a system without primary inputs. In practice, however, one wishes to perform model-checking on circuits that have inputs, and hence are not closed. To accomplish this goal, one combines a model of the subject circuit with an enclosing model of its environment.

The most general enclosing environment is an independent, nondeterministic assignment of zeros and ones to the input pins of the circuit on each clock edge. Many circuits, however, are designed to work correctly only under specific environmental assumptions. Thus, in practice, the enclosing environment must model the actual environment in which the circuit will operate.

There are at least two problems with the environment modeling approach. The construction of such environments can be a difficult and time-consuming procedure. Moreover, there is no clear methodology for ensuring that the environment model is a true abstraction of the actual environment in which the subject circuit will operate.

* Matt Kaufmann's current address is: EDS CIO Services, 98 San Jacinto Blvd. Suite 500, Austin, TX 78701

[1] PowerPC is a trademark of the International Business Machines Corporation, used under license therefrom.

Constraints, as implemented in Motorola's Verdict model-checker, address these two problems. They provide a simple way to model environments. Moreover, they support an assume/guarantee methodology for ensuring that the environment models created are conservative abstractions of the actual environment in which the circuit will operate. Constraints can be used easily as properties to monitor during simulation of large units.

A constraint is a boolean formula involving any signals occurring in a design, including inputs to the design. If a monitor is used, constraints may involve signals of the monitor. Conceptually, constraints can be described in three levels of generality. The simplest constraints involve only input signals. For example, a constraint that inputs A, B and C are one-hot is expressed by the following formula:

$$\texttt{(A} + \texttt{B} + \texttt{C)\&!(A\&B)\&!(A\&C)\&!(B\&C);} \qquad (1)$$

A second, more general, type of constraint involves signals that may depend upon the internal state of the design. A simple example from a microprocessor bus interface unit (BIU) is the assertion that if the BIU's address state machine is not in state "address idle" then its transaction-start input is not asserted, that is, if a transaction start input is asserted then the address state machine must be in the idle state:

$$\texttt{\$constraint(ts} \rightarrow \texttt{(addr\_state} = \texttt{`ADDR\_IDLE));} \qquad (2)$$

In this more general case of constraints, the inputs of the design, which determine the "next" state, may depend combinationally upon the current signals of the design but *only* on the current signals in the design, not past signals. It is a fundamental insight of the constraint approach that often a design under verification already contains sufficient information to determine what its input should be. Empirically, it was observed that environments being constructed replicated state already present in the design itself.

In the most general case of constraints, the inputs of a design depend not only on the current state of the design but also upon the history of reactions of the design to its inputs. In this case, it is necessary to instantiate a finite-state machine typically referred to as a *monitor* to "watch" the reaction of the design to its inputs and record information necessary to determine what the next input should be. Informally it is clear that, with the addition of monitors, constraints are as expressive as enclosing environment models.

One might imagine that by simply latching design inputs, one could just restrict model checking to the constrained state space. However, the constrained set of states may not be a Kripke model. In particular, it may be possible that some state s_0 is reachable from an initial constrained state through constrained states but such that s_0 does not have a next state that satisfies the constraints. It is our assumption that a state is valid if it is reachable from a valid input using constrained inputs for each step of the reachability calculation. If a "dead-end" state such as s_0 is encountered, this is evidence by itself that either the design or the constraints are incorrect.

This use of constraints has three key advantages over the use of enclosing environments. First, they are generally easier to write, and thus simplify the model-checking process. Second, a constraint, which is an assumption at one level of design hierarchy, automatically converts to an AG property, which is a proof obligation to verify at a higher level of hierarchy. For example, the BIU constraint converts directly into an AG property:

$$\mathtt{AG(ts} \rightarrow \mathtt{(addr\_state} = \mathtt{`ADDR\_IDLE));} \qquad (3)$$

Even in the case in which model checker capacity fails at a higher level, the constraint provides an invariant to monitor during simulation, the failure of which is evidence of faulty design or at least faulty specification. Thus, in a sense, use of constraints can be viewed as an assume/guarantee methodology. Finally, constraints document interface assumptions about design blocks in a set of simple formulas easily understood by designers. By contrast, environment modules constitute an awkward, unreadable, unmaintainable and unverifiable documentation of interface assumptions.

In [2], Long describes a very general method for doing compositional reasoning in the context of CTL model checking. In this work, Long presents a framework based on tableau construction. A procedure is given that constructs a model $T(\psi)$ for an arbitrary formula ψ. that allows one to verify properties of the form $< \psi > M < \phi >$, meaning that the composition of M with any environment satisfying ψ will satisfy ϕ, where M is an arbitrary Kripke structure, and ψ and ϕ are arbitrary ACTL formulas. ACTL is the subset of CTL in which all quantifiers are in essence universal.

The method involves the construction of a tableau, $T(\psi)$, that is a Kripke structure that represents the maximal environment that satisfies ψ. The composition $M||T(\psi)$ is then checked using standard model-checking algorithms.

The method presented here can be viewed as a special case of Long's work that is of practical interest. Instead of considering assumptions $< \psi >$ with the full generality of ACTL, we restrict attention to assumptions of the form $AG(P)$, where P is an elementary formula, i.e. a (boolean) formula free from temporal operators and path quantifiers. In such cases, as we shall show, it is possible to avoid constructing a tableau explicitly, so that no additional state is introduced into the model provided no state is introduced explicitly on behalf of a monitor.

2 The Methodology

2.1 Methodology basics

Constraints are combinational boolean properties specified by the user at the top level of the a design module, using the `$constraint` keyword. For example, the constraint below says that signals s1 and s2 cannot both be high. Verdict uses the Verilog expression language.

```
$constraint(!(s1 & s2));
```

Constraints that appear in the top-level module are called *assumptions.* Those that appear in instantiated modules are called *guarantees.*

- All model-checking is performed by considering only those computation paths that globally satisfy the assumptions specified by constraints in the top level module.
- In addition to checking the specifications supplied by the user, the model-checker will also check that all guarantees, specified by constraints in the instantiated modules, hold for all reachable states.

2.2 Assume/guarantee reasoning

Constraints form the basis for automated assume/guarantee reasoning. Suppose one wishes to verify a property, called m_1-spec, of a module m_1 that is instantiated inside a module m_0.

One first treats m_1 as the top-level design, using constraints within it to model an abstraction of its environment. Let these be called the m_1-constraints. Since m_1 is being treated as the top-level design, the m_1-constraints will be treated as assumptions. Specifications within m_1, the m_1-specs, will be verified only for those paths that globally satisfy these assumptions.

Next, one instantiates m_1 within the module m_0, treating m_0 as the top-level design. It is not necessary to re-verify the m_1-specs. Since m_1 is now an instantiated module, the m_1-constraints are treated as guarantees, not as assumptions. When verifying m_0, the model-checker will also verify that the m_1-constraints do, in fact, hold globally: for each constraint `C`, it checks the CTL formula `AG(C)`. In this way, the assumptions (m_1-constraints) that were made while verifying m_1 are discharged in the verification of m_0.

2.3 Monitors

Monitors allow constraints to specify sequential properties in addition to combinational properties. A monitor is generally a (Verilog) module instantiated within the top-level module of the design under test. Such a monitor has multiple inputs, and a single output. Conceptually, it monitors its inputs to ensure that they are behaving as expected. For example, it could monitor the input signals and selected state of the design under test to ensure that the environment that is driving them is following a given protocol. As long as things are behaving as expected, it continues to assert its output. However, if a protocol violation is detected its output is lowered.

By using the output of a monitor as the input to a constraint in the top-level module, a monitor can be used to restrict attention to state sequences that conform to the protocol. For example, a monitor within module m_1 could be used to restrict verification of the m_1-specs to the assumption that the inputs to m_1 have a certain sequential behavior.

When m_0 is subsequently verified, with m_1 instantiated within it, the same monitor can be used to discharge this assumption. The constraint that was previously treated as an assumption, is now treated as a guarantee. The model-checker will automatically check that the environment provided by m_0 fulfills this guarantee.

Of course, one can implement a monitor using in-line code, rather than an instantiated module, to achieve the same effect.

3 Semantics

A user view of constraints is explained above. Before giving the implementation we discuss the foundations.

Below, "constraint" refers to any constraint in the top-level module. Again, the constraints in other modules are turned into AG specs, automatically.

Notation. For a binary relation R and a set S. $Image(R, S)$ is the image of S by relation R, $\{\mathtt{t} \in \mathtt{range}(R) : \textit{for some } s \in S,\ R(s,t)\}$.

Definition. Fix a Kripke structure M with transition relation TR, and fix a Boolean constraint C. Let CI be the set of initial states of M that satisfy C; we assume that CI is not empty. Define M_C to be the Kripke structure obtained from M by replacing the initial state set with CI, and restricting the set of states to the set CR of states *reachable via* C, defined as follows. CR is the least set S of states containing CI such that for every pair of state $< s, s' >$ in TR for which $s \in S$ and s' satisfies C, then s' belongs to S. In other words, CR is the least fixed point of the following monotone functional F, where $C(S)$ is the set of states in S satisfying C:

$$F(S) = CI \cup [Image(TR, S) \cap C(S)].$$

The following describes the kind of structures appropriate for model-checking. The first restriction rules out vacuous models. The second restriction is standard for CTL semantics, and although it is automatically true for for traditional hardware models, it can of course fail in the presence of constraints.

Definition. A Kripke structure is *model-checkable* if (1) it has at least one initial state, and (2) every state has at least one successor state.

Our task at hand is as follows. We are given a Kripke structure M, at least implicitly, and a Boolean constraint C. In fact, what we have in hand is the transition relation derived from a given RTL or gate-level description, using for example Verilog or DSL.[2] We need to verify that M_C is model-checkable. If so, we want to do model-checking on M_C. In other words, the semantics of constraints (in the top module) are to cause the CTL model-checking to be done on the restricted Kripke structure described above, but with a check that model-checking is appropriate according to (1) and (2) above.

4 The Algorithm

There are two ways in which CTL formulas involving constraints can be evaluated: with forward reachability and without. For ease of explanation, consider

[2] DSL is an IBM proprietary RTL level description language.

the model in which all inputs are latched, i.e., assigned present-state ps and next-state ns BDD variables, in addition to the usual present- and next-state variables for the latches. In both cases, let $T(ps, ns)$ be the transition relation built without regard to constraints. It is assumed that the set CI of initial states is non-empty and that all its members satisfy the constraints `C`. (Actually, we form CI by intersecting the set of user-designated initial states with those that satisfy the constraint `C`.)

4.1 "Without Forward Reachability" Method

Using the full model M, evaluate the CTL formula `AG(C -> EX(C))`. If the formula is false, then the "without reachability" method fails and the user is notified to use the option to verify with reachability. The model-checker could have simply gone on using the reachability method at such a point, but users often do not notice warnings, and because reachability is expensive it seems safer to give the user the chance to review the situation before proceeding.

Otherwise, create a new transition relation whose range and domain satisfy constraints C. Let this new model be M_C. The rest of the evaluation is the same as Section 4.2 of the "with reachability" method below.

4.2 "With Forward Reachability" Method

Performing Forward Reachability Analysis Using the transition relation T evaluate the formula `EX(C)`. Perform forward reachability analysis with T in the following way. Each time a T-forward-image is calculated, the result is intersected with C. Check that the formula `EX(C)` holds for the set of newly created image states. If the check fails, Verdict quits with an error message about reaching a "dead-end" state. A feature that is not yet implemented would be to report a trace from an initial state satisfying `C`, through a set of states all satisfying `C` to a state that has no "next-state" satisfying `C`. This trace would show the user how the design can get into a "dead-end" state through valid transitions satisfying `C`.

Assuming that the `EX(C)`-check holds for each new frontier, the model checker has a set of states M_C in hand containing the initial states CI and such that for any state in M_C all of its T-next states satisfying C are in M_C.

Evaluating CTL Formulas Using M_C, evaluate the fair states and quit if there are no fair initial states. Modify the model again further restricting to fair states and check CTL specifications.

It should be pointed out that there are numerous variations and optimizations that can be made to the above algorithms. For example, one need not latch all of the inputs to create the transition relation T — just the inputs involved in the constraints and the CTL properties.

Furthermore, by existentially quantifying out inputs from C one can create C' which is the set of states which satisfy C for some input. Then one can cofactor each component of the relation T with C'. Likewise, one can cofactor the range of T as well.

It is important during model checking to make sure that all images and pre-images satisfy `C`, intersecting with the set of states satisfying `C` when necessary.

5 Examples

The use of constraints is illustrated by an example, derived from a PowerPC microprocessor design. The example is the controller for an instruction queue. The queue holds instructions that are waiting for operands, prior to being issued to one of several execution units. The controller performs the following tasks:

- When it has space, and instructions are available that are destined for the units served by this queue, it loads the instructions into the queue from the main instruction dispatch queue.
- When an instruction is ready for execution, and an execution unit is available, it issues the instruction to that execution unit.
- It tracks which queue entries have valid instructions.
- It tracks the relative age of instructions in the queue. This age information is used in the process of deciding whether an instruction is ready to execute.

The original module, as it existed in the PowerPC design, had several hundred primary inputs and outputs, and over a hundred internal latches. For the purpose of this presentation, the design has been simplified substantially by omitting some functionality, by shortening the queue, and by reducing the number of dispatch and issue ports. The clocking scheme has also been simplified from two-phase non-overlap, to a simple positive-edge-triggered synchronous design. However, the original design was indeed verified, and in spite of these simplifications, the basic verification issues remain unchanged from the original.

The module presented here is a queue with three entries, numbered 0, 1, and 2. It can receive up to two instructions from the main instruction dispatch queue per cycle, and issue up to two instructions, one to each of two execution units.

It has the following primary inputs

`clk` Internal latches update on the rising edge of this global clock.

`iq_loads[0 : 1]` A two bit input from the instruction dispatch queue. An asserted value indicates that data is available from the corresponding port.

`our_op[0 : 2]` Each bit indicates whether the corresponding instruction queue entry contains an operation that should be executed by one of the execution units serviced by this queue.

`exe0_ready` is asserted when by execution unit 0 when it is ready to receive a new instruction to execute.

`exe1_ready` is asserted by execution unit 1 when it is ready to receive a new instruction to execute.

`ops_ready[0 : 2]` Each bit is asserted when all the operands for the corresponding queue entry are available.

`flush[0 : 2]` For various reasons, such as branch mis-prediction, it is sometimes necessary to flush instructions from the queue. A queue entry will be flushed (by resetting its valid bit) when the corresponding flush input is asserted.

and the following primary outputs

`load0[0 : 1]` (resp. `load1[0 : 1]`, `load2[0 : 1]`) The controller asserts `load`i`[`j`]` to load queue element i from dispatch port j.

`issue0[0:2]` (resp. `issue1[0:2]`) The controller asserts `issue`i`[`j`]` to issue an instruction from queue element j to execution unit i.

`valid[0:2]` The controller asserts `valid[`i`]` when queue entry i has a valid entry.

The specifications that we wished to verify were straightforward. There were a number of safety specifications, of the form `AG(`p`)` where P is a (non-temporal) boolean formula expressed in terms of the inputs, outputs, and some internal signals of the module. Some were single cycle specifications, of the form `AG(P` $\rightarrow$ `AX Q)`. There were also three liveness specifications of the form `AG AF P`.

As it turned out, to verify even relatively simple properties about this design required some non-trivial assumptions about the design's environment. The environmental assumptions needed to verify the safety and single-cycle specifications were easily expressed using constraints that referenced only internal signals, primary inputs, and primary outputs. That is, no additional state was required. To verify the liveness specifications, however, it was necessary to introduce some additional state in order to constrain the inputs adequately.

One of the simplest constraints referred only to the primary inputs of the design under analysis. It simply required that no instructions be dispatched while a flush was in progress. Here and below, a vertical bar ("|") represents the or-reduction operator, true of a multi-bit signal when at least one bit is on.

$$\texttt{\$constraint}(\neg(|\texttt{flush}[0:2]\ \&\ |\texttt{iq\_loads}[0:1])) \tag{4}$$

Of primary concern to the designer of this block was the verification of the valid bits, and the age tracking mechanism. There are state bits, `q_age[0 : 2]` associated with age tracking. The bit, `q_age[0]` is asserted if entry 0 is older than entry 1. Bit `q_age[1]` is asserted if entry 0 is older than entry 2. Bit `q_age[2]` is asserted if entry 1 is older than entry 2. Should two entries arrive simultaneously, the entry with the smaller index is considered older. An empty queue slot is considered "newer" than an occupied queue slot.

The age bits were verified using several CTL specifications. Here we illustrate only age bit `q_age[0]`. The specifications for the other bits are completely analogous. Notice that in the case when one or both of `valid[0]` and `valid[1]` is de-asserted, the value of `q_age[0]` is fully determined by the values of `valid[0]` and `valid[1]`. This combinational part of the specification is captured by the following CTL formulas. Formula 5 gives the value of `q_age[0]` when `valid[0]` is low. Formula 6 gives the value when `valid[0]` is high but `valid[1]` is low.

$$\texttt{AG}(\neg\texttt{valid}[0] \rightarrow \neg\texttt{q\_age}[0]) \tag{5}$$

$$\texttt{AG}(\texttt{valid}[0]\ \&\ \neg\texttt{valid}[1] \rightarrow \texttt{q\_age}[0]) \tag{6}$$

When both valid[0] and valid[1] are asserted, the value of q_age depends on history. The following table shows only the possible transitions to a state in which valid[0] and valid[1] are both high. Transitions to states in which one or both of valid[0] and valid[1] are low are also possible, but the resulting value of q_age[0] in such a state has been determined by formulas 5 and 6 above.

Current State			Next State		
valid[0]	valid[1]	q_age[0]	valid[0]	valid[1]	q_age[0]
0	1	0	1	1	0
1	1	0	1	1	0
0	0	0	1	1	1
1	0	1	1	1	1
1	1	1	1	1	1

This set of transitions is expressed by the following two CTL formulas.

$$\begin{aligned}&\texttt{AG}((\texttt{valid}[1]\ \&\ \neg\texttt{q\_age}[0]) \rightarrow \\ &\quad \texttt{AX}((\texttt{valid}[0]\ \&\ \texttt{valid}[1]) \rightarrow \neg\texttt{q\_age}[0]))\end{aligned} \tag{7}$$

$$\begin{aligned}&\texttt{AG}((\neg\texttt{valid}[1] | \texttt{q\_age}[0]) \rightarrow \\ &\quad \texttt{AX}(\texttt{valid}[0]\ \&\ \texttt{valid}[1]) \rightarrow \texttt{q\_age}[0]))\end{aligned} \tag{8}$$

One might reasonably hope that the circuit would work correctly for all sequences of input stimulus but, unfortunately, this is not the case. In the chip environment, it turns out that a queue entry is never flushed unless all of the newer entries are flushed as well. The designer took advantage of this knowledge by omitting some (redundant under this assumption) flush information from the equations updating the age information. To verify this circuit, it is necessary to provide an environment that behaves according to these assumptions.

To accomplish this, the environment must know the relative age of the entries in the queue. One could, of course, construct an environment module that keeps track of this information. But to do so would add considerable additional state to the system. In our verification, we have capitalized on the fact that this information is already present *in the design under analysis*. Thus, we use the q_age bits themselves in the environment:

$$\texttt{\$constraint}(\neg(\texttt{flush}[0]\ \&\ \neg\texttt{flush}[1]\ \&\ \texttt{valid}[0]\ \&\ \texttt{valid}[1]\ \&\ \texttt{q\_age}[0])) \tag{9}$$

$$\texttt{\$constraint}(\neg(\texttt{flush}[1]\ \&\ \neg\texttt{flush}[0]\ \&\ \texttt{valid}[0]\ \&\ \texttt{valid}[1]\ \&\ \neg\texttt{q\_age}[0])) \tag{10}$$

On the surface, there might appear to be an alarming circularity in using the q_age bits to constrain the environment in order to verify the correctness of the q_age bits. However, the informal reasoning is valid. In this example, we are assuming the correctness of q_age *now* in order to establish the correctness of q_age *next cycle*. Note, however, that this (admittedly informal) reasoning would not be valid if q_age were combinationally dependent upon flush. For a more general treatment of this problem, in the context of abstraction, the reader is referred to [3]

Formally, however, the constraints do nothing more than restrict the scopes of path quantifiers in the CTL formulas to paths that globally satisfy the constraints. Correctness has thus been verified, under the assumption that sequences that violate the constraints will not occur. This assumption can be discharged formally, by model-checking the circuit within its enclosing environment, verifying the spec $AG(C)$, where C is the conjunction of the constraints above. In practise, this circuit was already at the limits of our model-checker's capacity. Nonetheless, the assumption can be validated informally, by checking that no constraint is violated during unit or full-chip simulation.

Finally, we wished to verify that, under certain fairness assumptions, every instruction queue element is eventually dispatched. These properties were easily expressed by the CTL specifications:

$$\texttt{AG AF}\neg\texttt{valid[0]} \tag{11}$$

$$\texttt{AG AF}\neg\texttt{valid[1]} \tag{12}$$

$$\texttt{AG AF}\neg\texttt{valid[2]} \tag{13}$$

As is typical with such specifications, these will only hold under certain fairness assumptions. In particular, we must assume that, for each queue element, there are both arguments and an execution unit available infinitely often. These assumptions alone, however, are not sufficient to verify the desired properties. The problem has to do with barriers. Some instructions are barriers, enforcing in-order execution. A barrier cannot be issued until all the instructions that precede it have completed. Furthermore, no instruction that follows a barrier can execute until the barrier has begun execution.

One could force the specifications to pass, by requiring that both execution units become free *simultaneously* infinitely often. However, this fairness constraint is really too strong. What is required is that once an execution unit becomes free, it does not become busy again until an instruction is issued to it.

To express this property using constraints requires some additional state, beyond that which is present explicitly in the design. The monitor must remember previous values, both of the `exe0_ready` and `exe1_ready` signals and of the `issue0` and `issue1` signals. Let `p_exe0_ready`, `p_exe1_ready`, `p_issue0`, and `p_issue1` be the previous values of the corresponding signals. Such values can be obtained easily in Verilog, by constructing a simple latch. The desired constraints are simply:

$$\texttt{\$constraint}(\neg(\neg\texttt{exe0\_ready}\ \&\ \texttt{p\_exe0\_ready}\ \&\ \neg\texttt{p\_issue0})) \tag{14}$$

That is, it should never be the case that

- execution unit 0 is not ready now,
- execution unit 0 was ready last cycle,
- and nothing was issued to unit 0 last cycle.

Similarly, it is necessary to constrain the signal `barrier`[0 : 2] so that each of its bits changes only when the corresponding queue entry is invalid — being a "barrier" is a static property of an instruction.

With these constraints, and the additional fairness constraints that require each execution unit (independently) to be available infinitely often, it was possible to establish the desired liveness properties.

6 Summary

Our approach to handling constraints is new in several ways.

- Constraints free the user from writing environment models to generate inputs.
- In many cases, constraints accomplish what environments accomplish with fewer BDD variables.
- Constraints allow the automated restriction of computation paths, for example using monitors. Because the restriction can depend on internal state, environment models alone cannot easily be used to accomplish this restriction.
- The algorithm handles input constraints that depend on the state of the design being verified.
- An assume/guarantee methodology allows both the assumption and verification of constraints. Constraints as verification properties can be used with conventional simulation validation.
- Constraints provide a convenient, easily understood method for documenting module interfaces, which can be used to catch errors during simulation as well as model checking.

References

1. E.M. Clarke and E.A. Emerson, ***Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic***, Proceedings of the Workshop on Logics of Programs, York town Heights, NY, Springer-Verlag LNCS no. 131, pp. 52 – 71, May 1981.
2. D.E. Long, "Model Checking, Abstraction, and Compositional Verification," School of Computer Science, Carnegie Mellon University publication CMU-CS-93-178, July 1993.
3. K. L. McMillan, "A Compositional Rule for Hardware Design Refinement," Orna Grunberg (Ed.) *Computer Aided Verification*, Proceedings of the 9th International Conference, Haifa, Israel, Springer-Verlag LNCS no. 1254, pp. 24 – 35, June 1997.