Centrum voor Wiskunde en Informatica

REPORTRAPPORT

SEN

Software Engineering

Software ENgineering

Bug hunting with false negatives

J.R. Calamé, N. Ioustinova, J.C. van de Pol, N. Sidorova

REPORT SEN-R0609 JUNE 2006

Centrum voor Wiskunde en Informatica (CWI) is the national research institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organisation for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

Copyright © 2006, Stichting Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Bug hunting with false negatives

ABSTRACT
Safe data abstractions are widely used for verification purposes. Positive verification results can be transferred to the concrete system. When a property is violated in the abstract system, one still has to check whether a concrete violation exists. However, even when the violation scenario is not reproducible in the concrete system (a false negative), it may still contain information on possible sources of bugs. Here we propose a bug hunting framework based on abstract violation scenarios. We first extract a violation pattern from an abstract violation scenario. The violation pattern represents multiple violation scenarios, increasing the chance that a corresponding concrete violation exists. Then we look for a concrete violation that corresponds to the violation pattern by using constraint solving techniques.

1998 ACM Computing Classification System: D.2.4; D.2.5
Keywords and Phrases: Data abstractions; verification; false negatives; debugging; abstract vs. concrete counterexamples

Bug Hunting with False Negatives

Jens R. Calamé[1], Natalia Ioustinova[1], Jaco van de Pol[1], Natalia Sidorova[2]
[1] CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
[2] Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

ABSTRACT
Safe data abstractions are widely used for verification purposes. Positive verification results can be transferred to the concrete system. When a property is violated in the abstract system, one still has to check whether a concrete violation exists. However, even when the violation scenario is not reproducible in the concrete system (a false negative), it may still contain information on possible sources of bugs.

Here we propose a bug hunting framework based on abstract violation scenarios. We first extract a violation pattern from an abstract violation scenario. The violation pattern represents multiple violation scenarios, increasing the chance that a corresponding concrete violation exists. Then we look for a concrete violation that corresponds to the violation pattern by using constraint solving techniques.

1998 ACM Computing Classification System: D.2.4 [Software/Program Verification], D.2.5 [Testing and Debugging].
Keywords and Phrases: Data abstractions, verification, false negatives, debugging, abstract vs. concrete counterexamples.

1. Introduction

Abstractions [6, 7, 8, 10, 12, 18] are widely used to reduce the state space of complex/parameterized/large systems for the verification purpose. Abstractions are often constructed in such a way that we can transfer positive verification results to the concrete model but not the negative ones, i.e. they are mostly used to check satisfaction rather than falsification of a property. Counterexamples found on the abstract system could have no counterpart in the concrete system. We further refer to this kind of counterexamples as *false negatives*.

In this paper we consider false negatives in the context of *data abstractions*, i.e. abstractions that substitute actual data values by abstract ones and operations on concrete data by operations on abstract data, depending on the property being verified. We use the timer abstraction from [9] as an illustrating example in this paper. This abstraction leaves all values of a discrete timer below k unchanged and maps all other values to the abstract value k^+. Note that the deterministic time progress operation *tick* (decreasing the values of active timers by one), becomes nondeterministic on the abstract model (see Fig. 1).

Suppose we use this timer abstraction to verify an LTL property mentioning actions a and b and obtain the trace $a.set(k^+).tick^3.b.a.set(k^+).tick^2.d)^\star$ as a counterexample for the abstract system. Not that the data abstraction influences the number of ticks the system may take. In case the number of consecutive *tick*s stays the same along any trace of the concrete system, the abstract trace cannot be reproduced on the concrete system and is thus a false negative.

Although the obtained counterexample is a false negative, it may still contain a clue for finding a concrete counterexample, e.g. in case the trace $a.set(k).tick^k.b.\ (a.set(k).tick^k.d)^\star$ is possible in the original system for some k. We can relax the found abstract counterexample by using the information that the operations on timers are influenced by the timer abstraction and check whether the concrete system contains a trace matching the pattern $a.any^\star.b.(a.any^\star.d)^\star$, where *any* represents any action on timers. Note that any such a trace would violate our property of interest. The pattern is more restrictive than the negation of the property, and moreover, it contains a cyclic part. Therefore, finding a trace of the concrete system satisfying the pattern is "easier" than a finding a trace of the concrete system violating the property when the enumerative model checking is concerned.

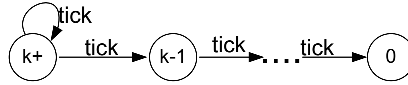

Figure 1: Abstracted timer

In this paper, we propose a framework that supports the bug hunting process described in the above example. Given an abstract counterexample (which is possibly a false negative) we relax actions influenced by the data abstraction and construct a *violation pattern* that represents a a set of traces violating the property and "similar" to the counterexample.

To check whether there is any concrete trace matching the violation pattern, we transform the violation pattern and the specification of the concrete system into a constraint program. Further a constraint solver is used to find a concrete trace matching the violation pattern, if such a trace exists. We tried out our approach by applying it for debugging *μCRL* specifications [1]. We use Eclipse Prolog [4] to implement constraint solving. Our first experiments showed that our framework can be successfully used for counter abstractions.

Related work First, we compare our method with the more traditional CEGAR approach (Counterexample-Guided Abstraction Refinement) [5, 17]. In both methods, abstractions preserve properties in one direction only: if the abstract system satisfies the property, so does the concrete system; a counterexample may however be a real one or a false negative (spurious). In the CEGAR method, a spurious counterexample is used to iteratively refine the abstraction; subsequently the model checking algorithm is called on the refined abstraction. Our method is to generalize the spurious counterexample, hoping to directly find some violation in the concrete specification, similar to the original spurious counterexample.

Note that in principle both methods can be combined: given a (possibly spurious) counterexample, one could search for a concrete violation using our method. If this is found, the CEGAR loop can be terminated early. If a concrete counterexample is not found, one can proceed to refine the abstraction, as in the CEGAR approach and iterate the verification process.

In [22], the authors address the problem of automating the analysis of counterexamples that have been produced when model checking the abstract model in order to determine whether they represent real system defects. They propose two techniques for this purpose: model checking on choose-free paths and abstract counterexample guided concrete simulation. In [21], an approach based on test generation is proposed for searching concrete instances of abstract counterexamples. The approach addresses only counterexamples for safety properties, i.e. works only for finite counterexamples. Unlike these approaches, we look for a concrete trace matching not a counterexample itself but a driven by it violation pattern.

Works orthogonal to our work are [15] and [23], where the authors propose model checking methods that rely on a refinement of an under-approximation of the system behaviour. These methods are aimed at the falsification of the desired property, and apply a refinement when a counterexample is not found. We aim at proving the property and try to find a concrete counterexample when we do not succeed to prove it.

The rest of the paper is organized as follows. We introduce some basic notions in Section 2 and data abstractions in Section 3. In Section 4, we present our framework . In Section 5 we describe our approach to finding concrete traces matching the violation pattern and discuss some experimental results. Finally, in Section 6 we summarise our contribution and discuss directions for further research.

2. PRELIMINARIES

Our operational model is based on synchronously communicating processes with top-level concurrency. This is a simplification of a model used in [24]. A specification *Spec* is given as the parallel composition $\Pi_{i=1}^{n} P_i$ of a finite number of processes. A process definition P is given by a four-tuple $(Var, Loc, \sigma_0, \rightarrow)$, where *Var* denotes a finite set of variables, and *Loc* denotes a finite set of *locations*, or control states. A mapping of variables to values is called a valuation; we denote the set of valuations by $Val = \{\eta \mid \eta \colon Var \rightarrow D\}$. We assume standard data domains such as $\mathbb{N}$, *Bool*, etc. We write D when leaving the data-domain unspecified and silently assume all expressions to be well-typed. Let $\Sigma = Loc \times Val$ be the set of states, where a process has one designated

initial state $\sigma_0 = (l_0, \eta_0) \in \Sigma$. The set $\rightarrow \subseteq Loc \times Act \times Loc$ denotes the set of edges. An *edge* describes changes of configurations specified by an *action* from a set *Act*.

As actions, we distinguish (1) *input* of a signal s containing a value to be assigned to a local variable, (2) *output* of a signal s together with a value described by an expression, and (3) internal *events*. Every action is *guarded* by a boolean expression g, its guard. The three classes of actions are written as $g \triangleright ?s(x)$, $g \triangleright !s(e)$, and $g \triangleright a, x := e$, respectively, and we use α, $\alpha' \ldots$ when leaving the class of actions unspecified. For an edge $(l, \alpha, \hat{l}) \in \rightarrow$, we write more suggestively $l \rightarrow_\alpha \hat{l}$.

Before we define the semantics, we introduce the notions of labeled transition system and of trace.

Definition 1 *(LTS).* A labeled transition system (*LTS*) is a tuple $M = (\Sigma, Act, \rightarrow, \sigma_{init})$ where $\Sigma \neq \emptyset$ is a set of states, *Act* is a set of labels (*actions*), $\rightarrow \subseteq \Sigma \times Act \times \Sigma$ is a transition relation, $\sigma_{init} \in \Sigma$ is the initial state.

Definition 2 *(Trace).* A *trace* β of M is a mapping $\beta : N \longrightarrow Act$, where $N = \{1, 2, \ldots, n\}$ or $N = \mathbb{N}$, for which there exists a mapping $\beta_\sigma : N \cup \{0\} \longrightarrow \Sigma$ such that $\beta_\sigma(i) \rightarrow_{\beta(i+1)} \beta_\sigma(i+1)$ for all $(i+1) \in N$. If $N = \mathbb{N}$, trace β is called an *infinite* trace; otherwise, it is a *finite* trace. The *length* $|\beta|$ of β is defined as $|N|$.

In the further text, we refer to the set of traces of M as $[\![M]\!]_{trace}$.

The step semantics is given by an *LTS* $M = (\Sigma, Lab, \rightarrow_\lambda, \sigma_0)$, where $\rightarrow_\lambda \subseteq \Sigma \times Lab \times \Sigma$ is given as a labelled transition relation between states. The labels differentiate internal event steps and communication steps, either input or output, which are labelled by a signal and a value being transmitted, i.e. $?s(v)$ or $!s(v)$, respectively.

Receiving a signal s with a communication parameter x, $l \rightarrow_{g \triangleright ?s(x)} \hat{l} \in \rightarrow$, results in updating the valuation $\eta_{[x \mapsto v]}$ according to the parameter of the signal and changing current location to $\hat{l}$. Output, $l \rightarrow_{g \triangleright !s(e)} \hat{l} \in \rightarrow$, is guarded, so sending a message involves evaluating the guard and the expression according to the current valuation. It leads to the change of the location of the process from l to $\hat{l}$.

Assignments, $l \rightarrow_{g \triangleright a, x := e} \hat{l} \in \rightarrow$, result in the change of a location and the update of the valuation $\eta_{[x \mapsto v]}$, where $[\![e]\!]_\eta = v$. Assignment transitions are labelled by the corresponding event label a.

Although we are working with specifications containing only one process definition, it does not limit our approach. Existing linearization techniques [14] allow to obtain a single process definition for a parallel composition of a finite number of process definitions by eliminating communication and parallel composition.

3. Safe Data Abstractions

Abstraction techniques are widely used to make the verification of complex/parameterised/infinite systems feasible. The main requirement for an abstraction is that the abstract system behavior should correctly reflect the behavior of the concrete system with respect to a verification task in the sense that (1) an abstraction should capture all essential points in the system behavior, i.e., be not "too abstract", and (2) an abstraction should be *safe*, which means that every property checked to be true on the abstract model, holds for the concrete one as well. This allows the transfer of positive verification results from the abstract model to the concrete one.

The concept of safe abstraction is well-developed within the *Abstract Interpretation* framework [7, 8, 10, 18]. Working within this framework guarantees the preservation (in the direction from the abstract to the concrete model) of the truth of formulas of temporal logics without existential quantification over paths, e.g. $\Box L_\mu^+$ (i.e., all formulas of the μ-calculus without negation and containing only the $\Box$ operator) or next-free LTL. Since the negative verification results are usually not preserved, counterexamples can be false negatives.

In practice, a data abstraction is usually applied directly on a system specification rather than on its semantics model. The requirement that Abstract Interpretation imposes on the relation between the concrete model and its safe abstraction can be formalized as a requirement on the relation between the data and the operations of the concrete system and their abstract counterparts as follows: Each value of the concrete domain Σ is mapped by a *description function* $\rho_d : \Sigma \longrightarrow {}_\alpha\Sigma$ to a value from the abstract domain ${}_\alpha\Sigma$. The abstract value "describes" the concrete value. We assume an ordering $\preceq$ on the abstract domain ${}_\alpha\Sigma$ according to the "precision" of abstract values: given a concrete value x and its abstract description $x^\alpha = \rho_d(x)$, we say that any $y^\alpha \in {}_\alpha\Sigma$ such that $x^\alpha \preceq y^\alpha$ is a *less precise* description of x.

For every operation (function) f on the concrete data domain, an abstract function f^α is defined, which "mimics" f. (For simplicity, we assume f to be a unary operation.) In general, the abstraction can be nondeterministic. This is formally captured by letting f^α be a function into the *powerset* over the domain of abstract

values. The requirement of mimicking is then formally phrased with the following *safety statement*:

$$\forall x \in \Sigma \; \exists y \in f^{\alpha}(\rho_d(x)) : \rho_d(f(x)) \preceq y. \tag{3.1}$$

A Timer Abstraction We use a timer abstraction as an illustrative example. The timer abstraction has been proposed in [3, 9]. The concept of timers is often used to specify time constraints imposed on a system or on a system environment. We use this concept in the context of discrete time where time is modelled as a digital logical clock and time progress is modelled by an action *tick* having the least priority in the system [2].

A timer can be either active or deactivated. An active timer keeps a value left until an expiration of the timer. An active timer with the value 0 expires. We model expiration of a timer t by a boolean guard g_t checking whether the value of the timer is 0. Time progression decreases values of active timers.

For a timer t, the concrete domain of timer values $\Sigma = \mathbb{N} \cup \{-1\}$, where -1 represents a deactivated timer, is replaced with the abstract domain ${}_{\alpha}\Sigma_t = \{-1, 0, \ldots, k_t - 1, k_t^+\}$, where the value k_t is a positive value defined by the user, assuming that the property we want to verify still holds even if we do not distinguish between the values of the timer greater than or equal to k_t. We overload the notation by using c $(-1 \le c < k_t)$ as an abstract value representing the single concrete value c, while c^+ describes the set of concrete values $\{c, c+1, c+2, \ldots\}$. We do not consider 0^+ abstraction here.

The description function ρ_t is defined as $\rho_t(c) = c$ if $c < k_t$ and $\rho_t(c) = k_t^+$ otherwise. Abstract operations on timers are defined in an intuitive way: setting a timer to value x becomes setting it to value $\rho_t(x)$; the timeout guard g_t^{α} is *true* iff $[\![t]\!] = 0$; and $tick^{\alpha}$ is a nondeterministic operation that changes the value of a timer from a to b according to the following rules: (1) if $a = -1$ then $b = -1$, (2) if $0 \le a < k_t$ then $b = a - 1$ (where "$-$" works on abstract values as on integers), (3) if $a = x^+$ then $b \in \{x^+, x - 1\}$. It is straightforward to show that the timer abstraction is safe. Applied to a concrete system it yields the safe abstract system [16].

4. Bug-Hunting With False Negatives

Here we provide an overview of our framework for bug hunting with false negatives. The bug hunting process comprises the following steps.

1. Specify a requirement as a formula ϕ of action-based next-free LTL [11].
2. Choose and apply a data abstraction to the specification of the concrete system and to the property.
3. Check whether the abstract system satisfies the abstract property.
   If the abstract system satisfies the property, so does the concrete system and the verification process stops. Otherwise, proceed with step 4.
4. Generate a violation pattern that specifies a set of traces "similar" to the counterexample and *violating* the property.
5. Check whether one of the traces of the concrete system matches the violation pattern.

Since steps 1-3 of the framework can be handled by existing data abstraction and model checking techniques, our contribution concerns steps 4-5 of the framework only.

4.1 Constructing a violation pattern

A counterexample that we obtain in case the property is violated on our abstract model is an infinite trace of the form $\beta_p \beta_s^w$ where β_p is a finite prefix and β_s^{ω} is a cyclic suffix with a finite *cycle base* β_s. For safety properties (claiming that "something bad" does not happen), the cycle base will be empty, while for liveness properties (stating that "something good" eventually happens) it is not empty.

Although β could have no counterpart in the concrete system, it may contain a clue about a counterexample present in the concrete system. Therefore we transform a counterexample β into a *violation pattern* V. We consider here *both* finite and infinite counterexamples.

A violation pattern is an *LTS* that accepts all traces leading to the distinguished state *Accept* that has no successor states. The violation pattern accepts only traces which are *"similar"* to the counterexample and

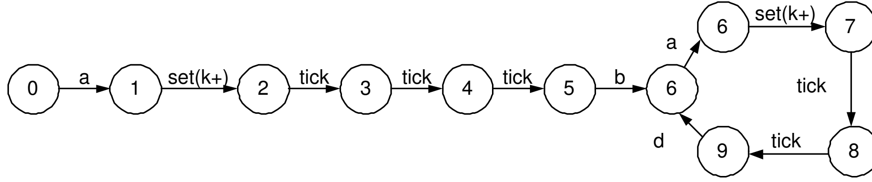

Figure 2: A counterexample

violate the property. The actions mentioned in the property and the information about the cyclic base of the counterexample are essential for the property violation. Therefore we keep this information in the violation pattern. For actions influenced by the abstraction, the order and the number of actions in a "similar" trace may differ from those of the counterexample.

First we will illustrate the idea of "similarity" on a simple example and then generalize it.

Assume that we model checked the property $\Box(a \longrightarrow \Diamond b)$ and obtained the following abstract counterexample (see Fig. 2): $a.set(k^{+}).tick^{3}.b.(a.set(k^{+}).tick^{2}.d)^{\omega}$. To guarantee that the property is violated by any trace accepted by the pattern, we keep *at least* actions a and b mentioned in the property as they are (see Fig. 3). Since we are searching for "similar" traces with an infinite *cyclic* suffix, we also need to keep information about *at least* one state of the cycle. Moreover, we specify when the search for a concrete counterexample can stop by providing a transition to the acceptance state. Here we do it by keeping the last action d of the suffix base (see Fig. 3). The actions *tick* and $set(k^{+})$ are not mentioned in the property and are definitely influenced by the timer abstraction. Therefore we "relax" these actions, meaning, we allow these actions to occur an arbitrary number of times in an arbitrary order (see states 1 and 3 of the violation pattern in Fig. 3).

We refer to the set of actions that do not want to relax as Act_{keep}. This set includes *at least* all the actions mentioned in the property. In the violation pattern, we distinguish a set of *cyclic* states that correspond to the states in the cyclic suffix. The last action of a finite counterexample leads to the accept state in the violation pattern. For an infinite counterexample, the last action of the cyclic base leads to the accept state.

Intuitively, we would like to relax only the actions influenced by the data abstraction (not necessarily all of them). These actions can be found by applying static analysis techniques. The more actions we keep, the faster we can check whether there is a concrete trace matching the pattern. By keeping too much, we may however end up with a violation pattern that specifies the traces having no counterparts in the concrete system. Formally:

Definition 3 *(Violation Pattern).* Given an abstract counterexample $\beta = \beta_p\beta_s^{\omega}$ and a set Act_{keep} of non-relaxed actions, a *violation pattern* is an extended *LTS* $V = (\Sigma,\ Act,\ T,\ \sigma_{init},\ C,\ Accept)$ constructed by Algorithm 4, where $C \subseteq \Sigma$ is a (possibly empty) set of cycle states and *Accept* is the finite state.

The set of finite traces ending in the accept state *Accept* is referred further as the set $[\![V]\!]_{atrace}$ of *accepted traces*. Traces which are not accepted by the pattern are referred further as a set $[\![V]\!]_{rtrace}$ of *refused traces*.

Given a counterexample $\beta = \beta_p\beta_s^{w}$ and a set Act_{keep} of actions to keep, Algorithm 4 constructs the violation pattern V. The algorithm starts with creating the initial state 0 of V and goes through $\beta_p\beta_s$. When the algorithm encounters an action to relax, it adds a self-loop transition labelled by this action to the current state of V. When it encounters an action to keep or reaches the end of the trace, it adds a transition from the current state to the

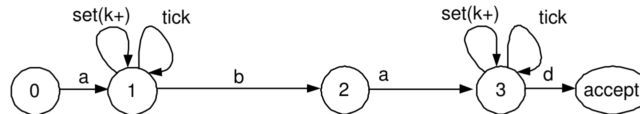

Figure 3: The violation pattern

Algorithm 1 *(Build Violation Pattern).*

1	**inputs** $\beta = \beta_p\beta_s^\omega, Act_{keep}$	trace, actions to keep
2	**output** $V = (\Sigma, Act, T, \sigma_{init}, C, Accept)$	violation pattern
3	$\sigma_{init} := 0$; $\Sigma := \{\sigma_{init}\}$;	initialization
4	$C := \emptyset$;	
5	$st := 0$;	current state st of V
6	**for all** $i = 1..\lvert\beta_p\beta_s\rvert$ **do**	for all steps of $\beta_p\beta_s$
7	{ **if** $\beta(i) \notin Act_{keep}$ **then** $T := T \cup \{(st, \beta(i), st)\}$;	add a relaxed step
8	**if** $\beta(i) \in Act_{keep}$ **or** $i = \lvert\beta_p\beta_s\rvert$ **then**	if a step to keep
9	{ $\Sigma := \Sigma \cup \{st+1\}$;	add a new state,
10	$T := T \cup \{(st, \beta(i), st+1)\}$;	add the step to the new state
11	$st := st + 1$;	proceed with the next state of V
12	}	
13	**if** $i = \lvert\beta_p\rvert + 1$ **then** $C := \{st\}$;	indicate the first state of the cycle
14	}	
15	$Accept := st$;	the last added state is the accept state

Figure 4: Algorithm constructing violation pattern

(new) next state labelled by this action. The set C of cyclic states is empty if we have a finite counterexample and contains the state corresponding to the first state of the cycle base for infinite counterexamples.

Clearly, all traces obtained from the traces of $[\![V]\!]_{atrace}$ by adding a loop going through C for infinite counterexamples violate the property ϕ.

4.2 Looking for a concrete counterexample

After we have constructed the violation pattern V, we check whether there is a concrete counterexample $\rho = \rho_p\rho_s^\omega$ such that $\rho_p\rho_s \in [\![V]\!]_{atrace}$.

For finite counterexamples (ρ_s is empty) in non-parameterised finite systems one can build a synchronous product of V and the concrete system and check whether the *Accept* state is reachable. For infinite counterexamples we additionally need to check that some state of ρ_σ corresponding to the *Accept* state coincides with a state of ρ_σ corresponding to C. We employ constraint solving [20] for finding a concrete counterexample, which allows us to check this additional condition for infinite traces, and also allows us to do it for infinite and parameterised systems.

To find a concrete trace matching the violation pattern V, we transform the specification of the concrete system and the violation pattern into a *constraint program* and formulate a *query* to find such a trace. Note that for a concrete system with an infinite state space, it is possible that the constraint solver will not terminate. Moreover, it is possible that the only traces that match the violation pattern are "spiral" traces, not "cycling" traces (i.e. we do have a loop with respect to actions, but some variable is e.g. infinitely growing) and we will not be able to find them. This limitation applies however to all known to us methods for identifying counterexamples.

The transformation of the specification of the concrete system into a rule system $\mathcal{RS}_{Spec}$ is defined in Table 1. Each edge of the specification *Spec* is mapped into a rule $A : -g$. In the rule, g is a guard and A is a *user defined* constraint of the form $name(state(l, \overline{Var}), state(\hat{l}, \overline{Var'}), param(Z))$. The first parameter *state* of the user defined constraint describes the source states corresponding to the edge in terms of control locations of a process and valuations of process variables. The second parameter *state* describes the destination states in terms of control locations of a process and valuations of process variables. The third parameter $param()$ contains parameters representing input and output values. The constraint is satisfied iff the guard *guard* is satisfied.

In ROUTPUT, the name of the constraint coincides with the signal s. Note that the values of the process variables $\overline{Var}$ remain unmodified and the output value is represented by the parameter Y whose value is given by the expression e. In RINPUT, the input leads to the substitution of the value of process variable x by the

$$\frac{l \to_{g \triangleright !s(e)} \hat{l} \in \to}{s(state(l,\overline{Var}), state(\hat{l},\overline{Var}), param(Y)) :- g \wedge Y = e.} \text{ ROUTPUT}$$

$$\frac{l \to_{g \triangleright ?s(x)} \hat{l} \in \to}{s(state(l,\overline{Var}), state(\hat{l},\overline{Var}_{[x \mapsto Y]}), param(Y)) :- g.} \text{ RINPUT}$$

$$\frac{l \to_{g \,\triangleright\, a, x:=e} \hat{l} \in \to}{a(state(l,\overline{Var}), state(\hat{l},\overline{Var}_{[x \mapsto e]}), param(_)) :- g.} \text{ RASSIGN}$$

Table 1: From specification *Spec* to rule system $\mathcal{RS}_{Spec}$

value of the input parameter Y. In RASSIGN, an assignment is represented by substituting the value of the process variable x by the valuation of the expression e. Event-rules have no local parameters, which is denoted by the underscore (*don't-care*) here.

The transformation of the edges of the violation pattern $V = (\Sigma,\ Act,\ T,\ \sigma_0,\ C,\ F)$ into the rules of the rule system $\mathcal{RS}_V$ is defined in Table 2. Note that we provide the transformation for input and output steps only. The transformation for the event steps is a straightforward simplification of the transformation for input and output steps. Intuitively, given a step of the violation pattern, a rule of $\mathcal{RS}_V$ checks whether the concrete system may make this step. The rules also take into account the information about cyclic states and the data abstraction.

Rules 1–4 of Table 2 transform the steps of the violation patterns into the rules of the form: $B :- A \wedge B' \wedge g_d \wedge g_c$. B is a user defined constraint of the form $\sigma(state(\vec{X}), O)$ specifying a pair of the source state $state(\vec{X})$ of the concrete system ($\vec{X}$ stands for $(l,\overline{Var})$) and the source state O of the violation pattern. B keeps the concrete states that are *possible* on a cycle in the set $\bar{C}$.

A is a user constraint of the form $a(state(\vec{X}), state(\vec{X}'), param(Y))$ as defined above. It represents a step on which the concrete system and the violation pattern can potentially synchronize.

B' is a user defined constraint that represents pairs of a destination state of the concrete system and a destination state of the violation pattern that can be reached by the step specified by A. The guard g_d checks whether the data parameter of the concrete action is a concretization of the data parameter of the abstract action. For the cyclic states, the guard g_c also checks whether the destination state of the concrete system has been visited already, i.e. whether it is in the set $\bar{C}$.

Rule 1 of Table 2 transforms input/output steps of the violation pattern whose destination state σ' is not the beginning of the cycle base and not the acceptance state. Therefore, we omit the cyclic guard g_c. The step specified by the $s(state(\vec{X}), state(\vec{X}'), param(Y))$ constraint leads to the change of the state to σ' in the violation

$$\frac{\sigma \to_{!s(v)} \hat{\sigma} \quad or \quad \sigma \to_{?s(v)} \hat{\sigma} \quad \hat{\sigma} \notin C \quad \hat{\sigma} \neq Accept}{\sigma(state(\vec{X}), \bar{C}) \ :- \ s(state(\vec{X}), state(\vec{X}'), param(Y)) \ \wedge Y \in \gamma(v) \ \wedge \ \hat{\sigma}(state(\vec{X}'), \bar{C})} \ (1)$$

$$\frac{\sigma \to_{!s(v)} \hat{\sigma} \quad or \quad \sigma \to_{?s(v)} \hat{\sigma} \quad C \neq \emptyset \quad \hat{\sigma} = Accept}{\sigma(state(\vec{X}), \bar{C}) \ :- \ s(state(\vec{X}), state(\vec{X}'), param(Y)) \ \wedge Y \in \gamma(v)} \ (2)$$

$$\frac{\sigma \to_{!s(v)} \hat{\sigma} \quad or \quad \sigma \to_{?s(v)} \hat{\sigma} \quad \hat{\sigma} \in C \quad \hat{\sigma} \neq Accept}{\sigma(state(\vec{X}), \bar{C}) \ :- \ s(state(\vec{X}), state(\vec{X}'), param(Y)) \ \wedge Y \in \gamma(v) \ \wedge \ \hat{\sigma}(state(\vec{X}'), [state(\vec{X}') \mid \bar{C}])} \ (3)$$

$$\frac{\sigma \to_{!s(v)} \hat{\sigma} \quad or \quad \sigma \to_{?s(v)} \hat{\sigma} \quad C \neq \emptyset \quad \hat{\sigma} = Accept}{\sigma(state(\vec{X}), \bar{C}) \ :- \ s(state(\vec{X}), state(\vec{X}'), param(Y)) \ \wedge Y \in \gamma(v) \ \wedge \ state(\vec{X}') \in \bar{C}} \ (4)$$

Table 2: From violation pattern VP to rule system $\mathcal{RS}_{VP}$

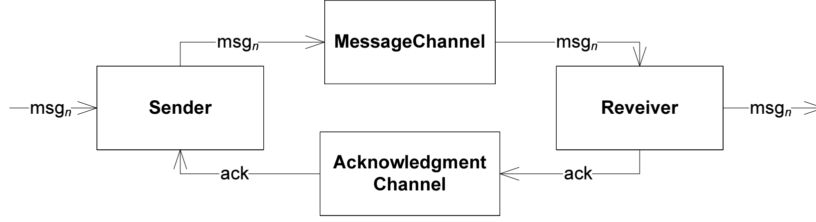

Figure 5: *PAR*Components

pattern and to the state $state(\vec{X}')$ in the concrete system. That is captured by the constraint $\hat{\sigma}(state(\vec{X}'),C)$ in the rule. The rule is satisfied only if both the violation pattern and the concrete system can make the specified input/output step and the action labelling the step of the concrete system satisfies the constraint $Y \in \gamma(v)$.

Rule 2 of Table 2 is applicable to the steps of the violation pattern built for a finite counterexample which lead to the acceptance state.

Rule 3 of Table 2 transforms the steps of the violation pattern leading the state which corresponds to the beginning of the cycle in case it does not coincide with the acceptance state. Since $\hat{\sigma}$ is the cyclic state, the pair of the state of the violation pattern and the state of the concrete system is *potentially cyclic*. By adding the constraint $\hat{\sigma}(state(\vec{X}'),[state(\vec{X}') \mid C])$, we add the concrete state $state(\vec{X}')$ satisfying the rule to the set $\bar{C}$ of states which are potentially cyclic.

Rule 4 of Table 2 transforms the step leading to the accept state of the violation pattern. In this case, we check whether the destination state $state(\vec{X}')$ of the concrete system has already been encountered before, i.e. we check whether it is in set $\bar{C}$ of potentially cyclic states.

The rule system $\mathcal{RS}_V$ together with the rule system $\mathcal{RS}_{Spec}$ form the constraint program. In order to check whether we can find a concrete counterexample matching the violation pattern, we transform the pair of the initial state of the violation pattern and the initial state of the concrete system into the query $\sigma_{init}(state(\vec{X}_{init}))$ and ask a constraint solver whether it has a solution in the constraint program formed by $\mathcal{RS}_{Spec}$ and $\mathcal{RS}_v$. If yes, it provides us a counterexample. Otherwise, we cannot give a conclusive answer and have to use e.g. abstraction refinement techniques to find out whether the property holds on the concrete system or not.

5. PAR

To check the applicability of our framework we performed a number of verification experiments with *µCRL* specifications. *µCRL* (micro Common Representation Language) is a specification language which is essentially an extension of the process algebra ACP with abstract data types and recursive definitions. The *µCRL* toolset provides tool support for analysis, abstraction, optimization and state space generation for *µCRL* specifications (see [13] for details about the language and [1] for details about the toolset). We use Eclipse Prolog [4] for constraint solving.

In this section we selected the positive acknowledgment retransmission protocol (PAR) [25] to illustrate the motivation for our work and our approach. The usual scenario for PAR includes a sender, a receiver, a message channel and an acknowledgment channel (see Fig.5). The channels delay the delivery of messages. Moreover, they can lose or corrupt messages. The sender receives a message from the upper layer, sends it to the receiver via the message channel, and waits for an acknowledgment from the receiver via the acknowledgment channel.

When the receiver has delivered the message to the upper layer it sends an acknowledgment to the sender. After the acknowledgment is received, the sender becomes ready to send a subsequent message. The receiver needs some time to deliver the received message to an upper layer. The sender handles lost messages by timing out. If the sender times out, it re-sends the message. The protocol works correctly iff the timer setting of the sender is larger than the sum of delays on communication channels.

Fig. 6 illustrates the idea of a well-known erroneous scenario for PAR. The sender times out while the acknowledgment is still on the way. The sender sends a duplicate, then receives the acknowledgment and believes that this is the acknowledgment for the duplicate. The sender sends the next message, which gets lost.

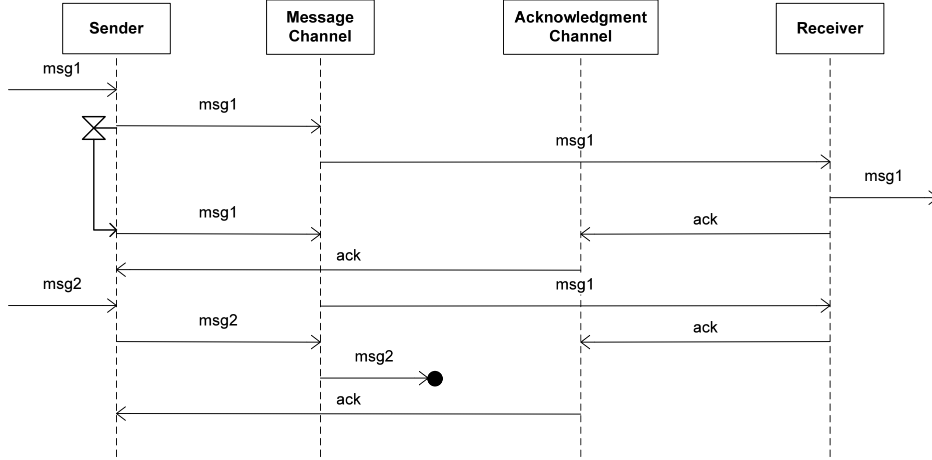

Figure 6: Counterexample for *PAR*

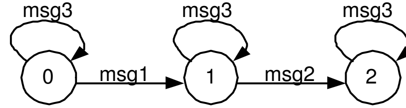

Figure 7: Environment for *PAR*

However, the sender receives the acknowledgment for the duplicate, which it believes to be the acknowledgment for the last message. Thus the sender does not retransmit the lost message and the protocol fails. To avoid this erroneous behavior, the timeout interval must be long enough to prevent a premature timeout, which means that the timeout interval should be larger than the sum of delays on the message channel, the acknowledgment channel and the receiver [25].

In order to perform the verification, we specified PAR in *μCRL* (see App. A and [2] for timed verification with *μCRL*). Model checkers usually cannot handle open systems. Therefore we also have specified an environment for PAR. In the environment, we differentiate only two messages *msg*1 and *msg*2, the rest is mapped to *msg*3. This environment is sufficient in order to check that no message loss, no duplicated delivery happens and that the messages are delivered in the same order as they are sent. The environment consists of two processes: one represents the upper layer for the sender and the other models the upper layer of the receiver. The behavior of the environment is illustrated in Fig 7. The process representing the upper layer of the sender sends an arbitrary number of *msg*3 to the sender, followed by sending *msg*1. Then it proceeds with sending an arbitrary number of *msg*3 followed by sending *msg*2 and finally sends an arbitrary number of *msg*3 again. The process representing the upper layer for the receiver receives the messages in the same fashion.

We tried to verify that for any setting of the sender timer exceeding some value k, all messages sent by the upper layer to the sender are eventually received by the upper layer from the receiver. To prove that the property holds for any setting of the sender timer exceeding k, we applied the timer abstraction described in Section 3 to the sender timer (see App. B for the abstract PAR specification). The property was not satisfied on the abstract system (since the k we took was less than the sum of the channel delays) and we obtained a counterexample (see App. C for the counterexample). The abstract counterexample was not reproducible on the concrete system, since the number of *tick* steps from a setting of the sender timer till its expiration varied along the trace due to the use of the abstraction.

We have transformed the abstract counterexample into the violation pattern (Fig. 8), where

$$
\begin{aligned}
A &= \{\mathsf{msg_out(msg1,F),tick,mdelay,msg_in(msg1,F),vfvs}\} \\
B &= \{\mathsf{ack_out,tick,adelay,retransmit,msg_out(msg1,F),ack_in}\} \\
C &= \{\mathsf{__mchbusy,tick,mdelay,msg_in(msg1,F),msg_out(msg2,T),vfws,ack_out,adelay,ack_in}\} \\
D &= \{\mathsf{mlose,msg_out(msg3,F),tick,mdelay,msg_in(msg3,F),vfws,ack_out,adelay}\} \\
E &= \{\mathsf{msg_out(msg3,T),tick,mdelay,msg_in(msg3,T),vfvs}\} \\
F &= \{\mathsf{ack_out,tick,adelay,ack_in}\} \\
G &= \{\mathsf{msg_out(msg3,F),tick,mdelay,msg_in(msg3,F),vfvs}\} \\
H &= \{\mathsf{ack_out,tick,adelay}\}
\end{aligned}
$$

The violation pattern basically says that after losing message *msg*2, the receiver will reject delivery of message *msg*3. Further the system gets into the loop where again and again message *msg*3 is delivered first with the sequence bit T and then with the sequence bit F. Thus, the message *msg*2 has never been delivered. In Fig. 8, this loop is entered in state 5 of the violation pattern.

We transform the violation pattern and the concrete specification of the system into resp. rule systems $\mathcal{RS}_V$ (see App. D) and $\mathcal{RS}_{Spec}$ (see App. E). With the Eclipse Prolog constraint solver, we obtain a solution for the query asking whether there any of the traces of the violation pattern has a concrete counterpart in the concrete system. The solution is indeed the counterexample for the property we have tried to verify (see App. F).

6. Conclusion

We proposed a novel framework for interpreting negative verification results obtained with the help of data abstractions. Existing approaches to handling abstract counterexamples try to find an exact counterpart of the counterexample (e.g. [22]). When no concrete counterpart can be found, the data abstraction is considered to be not fine enough and an abstraction refinement is applied (e.g. [5]).

In our framework we look for the useful information in false negatives. Given a specification of a system and a system property (formulated as an LTL-X formula), we first chose and apply a data abstraction to both of them and then verify the abstract property on the abstract system. If the verification results in a violation of the abstract property and the obtained counterexample has no counterpart in the concrete system, we transform the counterexample into a violation pattern, which is further used to guide the search for concrete counterexamples.

The framework allows to handle not only counterexamples obtained when verifying safety properties but also for liveness properties. Moreover, it can be applied for searching concrete counterexamples in parameterized and infinite state systems. Success is not always guaranteed – the violation pattern can be too strict, concrete counterexamples can have a "spiral" form, or there could be no counterexample at all since the property just holds on the concrete system. Still, our approach can help in finding counterexamples in those cases when a data abstraction influences the order and the number of some actions, e.g. as counter abstractions do.

The approach to generate a violation pattern leaves a certain freedom in the sense that the set of actions to relax can be more/less restrictive. Tuning the violation pattern or using the expertise of system developers to pick an appropriate set of actions to relax can be potentially less costly than repeating the abstraction/refinement cycle immediately. More case studies comparing both approaches and trying combinations are still needed.

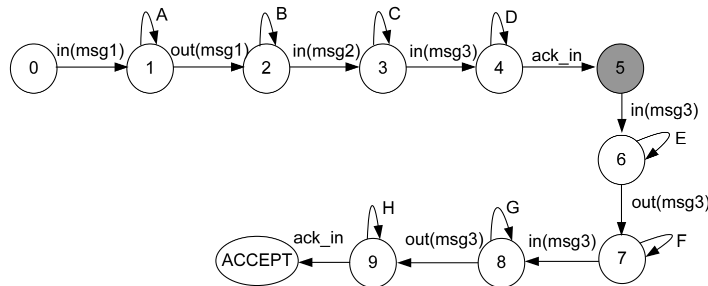

Figure 8: Violation Pattern for PAR

References

1. S. C. C. Blom, W. J. Fokkink, J. F. Groote, I. A. van Langevelde, B. Lisser, and J. C. van de Pol. μCRL: a toolset for analysing algebraic specifications. In G. Berry, H. Comon, and A. Finkel, editors, *13th International Conference on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 250–254. Springer-Verlag, 2001.
2. S. C. C. Blom, N. Ioustinova, and N. Sidorova. Timed verification with μCRL. In M. Broy and A. Zamulin, editors, *Proc. of the 5th Int. Conf. Perspectives of System Informatics*, volume 2890 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 2003.
3. D. Bošnački, N. Ioustinova, and N. Sidorova. Using fairness to make abstractions work. In S. Graf and L. Mounier, editors, *Proc. of the 11th Int. Spin Workshop on Model Checking of Software*, volume 2989 of *Lecture Notes in Computer Science*, pages 198–215. Springer, 2004.
4. P. Brisset, H. E. Sakkout, T. Frühwirth, C. Gervet, W. Harvey, M. Meier, S. Novello, T. L. Provost, J. Schimpf, K. Shen, and M. Wallace. *ECLIPSe Constraint Library Manual*, version 5.9 edition, May 2006. `http://eclipse.crosscoreop.com/eclipse/doc/libman.pdf`.
5. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
6. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.
7. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM Press.
8. D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD dissertation, Eindhoven University of Technology, July 1996.
9. D. Dams and R. Gerth. The bounded retransmission protocol revisited. *Electronic Notes in Theoretical Computer Science*, 9:26, 1999.
10. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2):253–291, 1997.
11. D. Giannakopoulou. *Model Checking for Concurrent Software Architectures*. PhD thesis, Imperial College of Science Technology and Medicine, University of London, March 1999.

12. S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In O. Grumberg, editor, *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.
13. J. F. Groote. The syntax and semantics of timed μCRL. SEN R9709, CWI, Amsterdam, 1997.
14. J. F. Groote, A. Ponse, and Y. S. Usenko. Linearization in parallel pCRL. *Journal of Logic and Algebraic Programming*, 48(1-2):39–72, 2001.
15. O. Grumberg, F. Lerda, O. Strichman, and M. Theobald. Proof-guided underapproximation-widening for multi-process systems. In J. Palsberg and M. Abadi, editors, *POPL*, pages 122–131. ACM, 2005.
16. N. Ioustinova. *Abstractions and Static Analysis for Verifying Reactive Systems*. PhD thesis, Free University of Amsterdam, 2004.
17. Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In Margaria and Yi [19], pages 98–112.
18. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.
19. T. Margaria and W. Yi, editors. *TACAS 2001*, volume 2031 of *Lecture Notes in Computer Science*. Springer, 2001.
20. K. Marriott and P. J. Stuckey. *Programming with Constraints – An Introduction*. MIT Press, Cambridge, 1998.
21. G. Pace, N. Halbwachs, and P. Raymond. Counter-example generation in symbolic abstract model-checking. *Int. J. Softw. Tools Technol. Transf.*, 5(2):158–164, 2004.
22. C. S. Pasareanu, M. B. Dwyer, and W. Visser. Finding feasible counter-examples when model checking abstracted java programs. In Margaria and Yi [19], pages 284–298.
23. C. S. Pasareanu, R. Pelánek, and W. Visser. Concrete model checking with abstract matching and refinement. In K. Etessami and S. K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 2005.
24. N. Sidorova and M. Steffen. Embedding chaos. In P. Cousot, editor, *Proc. 8th International Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, pages 319–334. Springer-Verlag, 2001.
25. A. S. Tanenbaum. *Computer Networks*. Prentice Hall International, Inc., 1981.

A. μCRL Specification of PAR

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  sort  Bool (Booleans from ysu)                             %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort Bool
func
  T,F: -> Bool
map
  and: Bool#Bool      -> Bool
  or:  Bool#Bool      -> Bool
  not: Bool           -> Bool
  if:  Bool#Bool#Bool -> Bool
  eq:  Bool#Bool      -> Bool
  gt:  Bool#Bool      -> Bool
var
  b,b1,b2: Bool
rew
  and(T,b)=b                     and(b,T)=b
  and(b,F)=F                     and(F,b)=F
  and(b,b)=b
  and(b,not(b))=F                and(not(b),b)=F
  and(or(b,b1),b2)=or(and(b,b2),and(b1,b2))
  and(b,or(b1,b2))=or(and(b,b1),and(b,b2))

  or(T,b)=T                      or(b,T)=T
  or(b,F)=b                      or(F,b)=b
  or(b,b)=b
  or(b,not(b))=T                 or(not(b),b)=T

  not(F)=T                       not(T)=F
  not(not(b))=b
  not(or(b,b1))=and(not(b),not(b1))
  not(and(b,b1))=or(not(b),not(b1))

  if(T,b1,b2)=b1                 if(F,b1,b2)=b2
  if(b,b1,b1)=b1                 if(not(b),b1,b2)=if(b,b2,b1)

  eq(b, b)=T
  eq(T, F)=F
  eq(F, T)=F

  gt(b,b)=F gt(T,F)=T gt(b,T)=F

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%  sort Nat (Natural numbers with nonbinary representation)   %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort Nat
func
  0:                 -> Nat
  S:     Nat -> Nat       % 2n+1
map
  eq:            Nat#Nat         -> Bool

  succ:          Nat             -> Nat  % n+1
  pred:          Nat             -> Nat  % n-1 (partial)
  gt:            Nat#Nat         -> Bool % greater than
  if:            Bool#Nat#Nat    -> Nat
var
  n,m: Nat b:Bool
rew
  gt(n,n)=F gt(0,n)=F gt(S(n),0)=T gt(S(n),S(m))=gt(n,m)

  eq(n,n)=T
```

```
  eq(S(n),0)=F
  eq(0,S(n))=F
  eq(S(n),S(m))=eq(n,m)

  succ(0)=S(0)                  % 0+1=2*0+1
  succ(n)=S(n)                  % (2n+1)+1=2n+2

  pred(S(n))=n                  % (2n+1)-1=2n+2
  pred(0)=0                         % (2n+2)-1=2n+1

  if(T,n,m)=n if(F,n,m)=m if(b,n,n)=n if(not(b),n,m)=if(b,m,n)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% sort Msg (messages mod 3)                                    %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort Msg
func
  msg1:->Msg
  msg2:->Msg
  msg3:->Msg
map
  next: Msg -> Msg
  intonat: Msg -> Nat
  eq:Msg#Msg -> Bool
var n, m:Msg
rew
  next(msg1) = msg2
  next(msg2) = msg3
  next(msg3) = msg1
  intonat(msg1) = S(0)
  intonat(msg2) = S(S(0))
  intonat(msg3) = S(S(S(0)))

  eq(n,m) = eq(intonat(n), intonat(m))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Timer sort (upper bound is 9)                               %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort Timer
func
  off:-> Timer
  on:Nat->Timer
map
  pred:Timer->Timer
  expire:Timer->Bool
var t:Timer n:Nat
rew
  expire(off)=F
  expire(on(n))=eq(0,n)
  pred(on(n))=on(pred(n))
  pred(off)=off

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% actions and communication                                   %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
act
mchbusy _mchbusy __mchbusy
  udelay
  retransmit SSf
  vfvs
  vfws
  wfws
  mdelay mlose
```

```
    adelay alose
    delay
    err
130 in:Msg _in:Msg __in:Msg
    out:Msg  _out:Msg __out:Msg
    tick _tick
    ack_out_rcv ack_in_ch ack_out_ch ack_in_snd ack_in ack_out
    msg_out_snd:Msg#Bool msg_in_ch:Msg#Bool msg_out_ch:Msg#Bool msg_in_rcv:Msg#Bool
135 msg_in:Msg#Bool
    msg_out:Msg#Bool

    ticka tickb

140 comm
    mchbusy|_mchbusy = __mchbusy
    tick|tick =_tick
    in| _in = __in
    out| _out = __out
145 ack_out_rcv | ack_in_ch = ack_out
    ack_out_ch  | ack_in_snd = ack_in
    msg_out_snd | msg_in_ch = msg_out
    msg_out_ch  | msg_in_rcv = msg_in

150 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%% process sender                                             %%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    proc Sender_Rh(sc:Timer, sn:Bool)=
      udelay.Sender_Rh(on(S(0)),sn)<|expire(sc)|>delta
155   +
      sum(m:Msg, _in(m).Sender_Sf(m,sn)<|expire(sc)|>delta)
      +
      tick.Sender_Rh(pred(sc),sn)<|not(expire(sc))|>delta

160 proc Sender_Sf(msg:Msg,  sn:Bool)=
      msg_out_snd(msg, sn).Sender_Ws(on(S(S(S(S(0))))),msg,sn)
      +
      mchbusy.Sender_Waiting_4Mch(on(S(0)), msg, sn)

165 proc Sender_Waiting_4Mch(t:Timer, msg:Msg, sn:Bool)=
      Sender_Sf(msg,  sn) <|expire(t)|>delta
      +
      tick.Sender_Waiting_4Mch(pred(t), msg, sn)<|not(expire(t))|>delta

170 proc Sender_Ws(sc:Timer, msg:Msg, sn:Bool)=
      ack_in_snd.Sender_Rh(on(0), not(sn))
      +
      retransmit.Sender_Sf(msg, sn)<|expire(sc)|>delta
      +
175   tick.Sender_Ws(pred(sc),msg,sn)<|not(expire(sc))|>delta

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%% process msg channel                                        %%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
180 proc Msg_Chan(mrc:Timer)=
      sum(m:Msg, sum(b:Bool, msg_in_ch(m, b).Msg_Chan_W(on(S(S(0))),m, b)))
      +
      tick.Msg_Chan(off)<|not(expire(mrc))|>delta

185
    proc Msg_Chan_W(mrc:Timer, msg:Msg, b:Bool)=
      mdelay.Msg_Chan_D(off, msg,b)<|expire(mrc)|>delta
      +
      _mchbusy.Msg_Chan_W(mrc,msg, b)<|not(expire(mrc))|>delta
190   +
```

```
  tick.(Msg_Chan_W(pred(mrc),msg, b))<|not(expire(mrc))|>delta


proc Msg_Chan_D(mrc:Timer, msg:Msg,  b:Bool)=
  msg_out_ch(msg, b).Msg_Chan(off)
  +
  mlose.Msg_Chan(off)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% process receiver                                             %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc Receiver_Wf(esn:Bool)=
  sum(m:Msg, sum(b:Bool, msg_in_rcv(m, b).Receiver_Wf1(m, b, esn)))
  +
  tick.Receiver_Wf(esn)

proc Receiver_Wf1(rm:Msg,  rsn:Bool, esn:Bool)=
  vfvs.Receiver_Sa1( rsn, rm,esn)<|eq(rsn, esn)|>delta
  +
  vfws.Receiver_Sa(esn)<|not(eq(rsn, esn))|>delta

proc Receiver_Sa1(rsn:Bool, rm:Msg, esn:Bool)=
  _out(rm).Receiver_Sa(not(esn))

proc Receiver_Sa(esn:Bool)=
  ack_out_rcv.Receiver_Wf(esn)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% process ack channel                                          %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc Ack_Chan(arc:Timer)=
  ack_in_ch.Ack_Chan_W(on(S(S(0))))
  +
  tick.Ack_Chan(off)<|not(expire(arc))|>delta

proc Ack_Chan_W(arc:Timer)=
  adelay.Ack_Chan_D(off)<|expire(arc)|>delta
  +
  tick.(Ack_Chan_W(pred(arc)))<|not(expire(arc))|>delta

proc Ack_Chan_D(arc:Timer)=
  ack_out_ch.Ack_Chan(off)
  +
  alose.Ack_Chan(off)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% process environment                                          %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc Env1(had1:Bool,had2:Bool) =
  in(msg1).Env1(T,F)<|and(eq(had1,F),eq(had2,F))|>delta
  +
  in(msg2).Env1(T,T)<|and(eq(had1,T),eq(had2,F))|>delta
  +
  in(msg3).Env1(had1,had2)

proc Env2(had1:Bool,had2:Bool) =
  out(msg1).Env2(T,F)<|and(eq(had1,F),eq(had2,F))|>delta
  +
  out(msg2).Env2(T,T)<|and(eq(had1,T),eq(had2,F))|>delta
  +
  out(msg3).Env2(had1,had2)

init(
 encap({_in, in, out, _out},
```

```
255  rename({_tick->tick},
     encap({tick, msg_out_ch, msg_in_rcv, ack_out_ch, ack_in_snd},
     rename({_tick->tick},encap({tick, msg_in_ch,msg_out_snd, mchbusy,_mchbusy}, Sender_Rh(on
         (0),F) || Msg_Chan(off)))
     ||
     rename({_tick->tick},encap({tick, ack_in_ch, ack_out_rcv}, Receiver_Wf(F) || Ack_Chan(off)
         ))
260  )
     )
           || Env1(F,F) || Env2(F,F)
     )
    )
```

B. Abstracted *μCRL* Specification of PAR

```
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%  sort  Bool (Booleans from ysu)                            %%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    sort Bool
 5  func
      T,F: -> Bool
    map
      and: Bool#Bool      -> Bool
      or:  Bool#Bool      -> Bool
10    not: Bool           -> Bool
      if:  Bool#Bool#Bool -> Bool
      eq:  Bool#Bool      -> Bool
      gt:  Bool#Bool      -> Bool
    var
15    b,b1,b2: Bool
    rew
      and(T,b)=b                    and(b,T)=b
      and(b,F)=F                    and(F,b)=F
      and(b,b)=b
20    and(b,not(b))=F               and(not(b),b)=F
      and(or(b,b1),b2)=or(and(b,b2),and(b1,b2))
      and(b,or(b1,b2))=or(and(b,b1),and(b,b2))

      or(T,b)=T                     or(b,T)=T
25    or(b,F)=b                     or(F,b)=b
      or(b,b)=b
      or(b,not(b))=T                or(not(b),b)=T

      not(F)=T                      not(T)=F
30    not(not(b))=b
      not(or(b,b1))=and(not(b),not(b1))
      not(and(b,b1))=or(not(b),not(b1))

      if(T,b1,b2)=b1                if(F,b1,b2)=b2
35    if(b,b1,b1)=b1                if(not(b),b1,b2)=if(b,b2,b1)

       eq(b, b)=T
       eq(T, F)=F
       eq(F, T)=F
40
    gt(b,b)=F gt(T,F)=T gt(b,T)=F

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%  sort Nat (Natural numbers with binary representation [ysu])%%%
45  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    sort Nat
    func
      0:                -> Nat
      x2p1:         Nat -> Nat      % 2n+1
50    x2p2:         Nat -> Nat      % 2n+2
```

```
map
  eq:            Nat#Nat          -> Bool

  1,2,3,4,5,6,7,8,9,10,11,12,13,14,
  15,16,17,18,19,20,21,22,23,24,25,
  26,27,28,29,30,31,32,33,34,35,36,
  37,38:                          -> Nat  % useful abbreviations

  x2p0:          Nat              -> Nat  % 2n
  succ:          Nat              -> Nat  % n+1
  pred:          Nat              -> Nat  % n-1 (partial)
  gt:            Nat#Nat          -> Bool % greater than
  if:            Bool#Nat#Nat     -> Nat
var
  n,m: Nat b:Bool
rew
  gt(n,n)=F gt(0,n)=F gt(x2p1(n),0)=T gt(x2p2(n),0)=T

  gt(x2p1(n),x2p2(m))=gt(n,m)
  gt(x2p2(n),x2p1(m))=not(gt(m,n))

  gt(x2p1(n),x2p1(m))=gt(n,m)
  gt(x2p2(n),x2p2(m))=gt(n,m)

  eq(n,n)=T
  eq(x2p1(n),0)=F
  eq(0,x2p1(n))=F
  eq(x2p2(n),0)=F
  eq(0,x2p2(n))=F
  eq(x2p1(n),x2p2(m))=F
  eq(x2p2(n),x2p1(m))=F
  eq(x2p2(n),x2p2(m))=eq(n,m)
  eq(x2p1(n),x2p1(m))=eq(n,m)

   1=x2p1(0)     2=x2p2(0)                          % 1=2*0+1  2=2*0+2
   3=x2p1(1)     4=x2p2(1)                          % 3=2*1+1  4=2*1+2
   5=x2p1(2)     6=x2p2(2)                          % 5=2*2+1  6=2*2+2
   7=x2p1(3)     8=x2p2(3)
   9=x2p1(4)    10=x2p2(4)
  11=x2p1(5)    12=x2p2(5)
  13=x2p1(6)    14=x2p2(6)
  15=x2p1(7)    16=x2p2(7)
  17=x2p1(8)    18=x2p2(8)
  19=x2p1(9)    20=x2p2(9)     29=x2p1(14)  30=x2p2(14)
  21=x2p1(10)   22=x2p2(10)    31=x2p1(15)  32=x2p2(15)
  23=x2p1(11)   24=x2p2(11)    33=x2p1(16)  34=x2p2(16)
  25=x2p1(12)   26=x2p2(12)    35=x2p1(17)  36=x2p2(17)
  27=x2p1(13)   28=x2p2(13)    37=x2p1(18)  38=x2p2(18)

  x2p0(0)=0                             % 2*0=0
  x2p0(x2p1(n))=x2p2(x2p0(n))           % 2(2n+1)=2(2n)+2
  x2p0(x2p2(n))=x2p2(x2p1(n))           % 2(2n+2)=2((2n+1)+1)=2(2n+1)+2

  succ(0)=x2p1(0)                       % 0+1=2*0+1
  succ(x2p1(n))=x2p2(n)                 % (2n+1)+1=2n+2
  succ(x2p2(n))=x2p1(succ(n))           % (2n+2)+1=2(n+1)+1

  pred(x2p1(n))=x2p0(n)                 % (2n+1)-1=2n+2
  pred(x2p2(n))=x2p1(n)                 % (2n+2)-1=2n+1

  if(T,n,m)=n if(F,n,m)=m if(b,n,n)=n if(not(b),n,m)=if(b,m,n)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
115 %%% sort Msg (messages mod 3)                                        %%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    sort Msg
    func
      msg1:->Msg
120   msg2:->Msg
      msg3:->Msg
    map
      next: Msg -> Msg
      intonat: Msg -> Nat
125   eq:Msg#Msg -> Bool
    var n, m:Msg
    rew
      next(msg1) = msg2
      next(msg2) = msg3
130   next(msg3) = msg1
      intonat(msg1) = 1
      intonat(msg2) = 2
      intonat(msg3) = 3

135 eq(n,m) = eq(intonat(n), intonat(m))

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%% Timer sort (upper bound is 9)                                    %%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
140 sort Timer
    func
      off:-> Timer
      on:Nat->Timer
    map
145   pred:Timer->Timer
      expire:Timer->Bool
    var t:Timer n:Nat
    rew
      expire(off)=F
150   expire(on(n))=eq(0,n)
      pred(on(n))=on(pred(n))
      pred(off)=off

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
155 %%% Abstracted Timer sort                                            %%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    sort Timer_abstr
    func
      known:Timer-> Timer_abstr
160   kplus:->Timer_abstr
    map
      pred:Timer_abstr->Timer_abstr
      preda:Timer_abstr->Timer_abstr
      expire:Timer_abstr->Bool
165 var t:Timer n:Nat
    rew
          expire(known(off))=F
          expire(known(on(n)))=eq(0,n)
          expire(kplus)=F
170
    % Simulation of nondeterminism on level of rewrite rules.
          pred(known(on(n)))=known(on(pred(n)))
          pred(known(off))=known(off)
          pred(kplus)=known(on(3)) % for k=3, adjust for other values of k
175
          preda(known(on(n)))=known(on(pred(n)))
          preda(known(off))=known(off)
          preda(kplus)=kplus
```

```
180 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%% actions and communication                                   %%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    act
    mchbusy _mchbusy __mchbusy
185   udelay
      retransmit SSf
      vfvs
      vfws
      wfws
190   mdelay mlose
      adelay alose
      delay
      err
      in:Msg _in:Msg __in:Msg
195   out:Msg  _out:Msg __out:Msg
      tick _tick
      ack_out_rcv ack_in_ch ack_out_ch ack_in_snd ack_in ack_out
      msg_out_snd:Msg#Bool msg_in_ch:Msg#Bool msg_out_ch:Msg#Bool msg_in_rcv:Msg#Bool
      msg_in:Msg#Bool
200   msg_out:Msg#Bool

      ticka tickb

    comm
205   mchbusy|_mchbusy = __mchbusy
      tick|tick =_tick
      in| _in = __in
      out| _out = __out
      ack_out_rcv | ack_in_ch = ack_out
210   ack_out_ch  | ack_in_snd = ack_in
      msg_out_snd | msg_in_ch = msg_out
      msg_out_ch  | msg_in_rcv = msg_in

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
215 %%% process sender                                              %%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    proc Sender_Rh(sc:Timer_abstr, sn:Bool)=
       udelay.Sender_Rh(known(on(1)),sn)<|expire(sc)|>delta
       +
220    sum(m:Msg, _in(m).Sender_Sf(m,sn)<|expire(sc)|>delta)
       +
       tick.Sender_Rh(preda(sc),sn)<|not(expire(sc))|>delta
        +
       tick.Sender_Rh(pred(sc),sn)<|not(expire(sc))|>delta
225
    proc Sender_Sf(msg:Msg,  sn:Bool)=
       msg_out_snd(msg, sn).Sender_Ws(kplus,msg,sn)
        +
       mchbusy.Sender_Waiting_4Mch(on(1), msg, sn)
230
    proc Sender_Waiting_4Mch(t:Timer, msg:Msg, sn:Bool)=
         Sender_Sf(msg,  sn) <|expire(t)|>delta
          +
          tick.Sender_Waiting_4Mch(pred(t), msg, sn)<|not(expire(t))|>delta
235
    proc Sender_Ws(sc:Timer_abstr, msg:Msg, sn:Bool)=
       ack_in_snd.Sender_Rh(known(on(0)), not(sn))
       +
       retransmit.Sender_Sf(msg, sn)<|expire(sc)|>delta
240    +
       tick.Sender_Ws(pred(sc),msg,sn)<|not(expire(sc))|>delta
       +
```

```
    tick.Sender_Ws(preda(sc),msg,sn)<|not(expire(sc))|>delta

245 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%% process msg channel                                       %%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    proc Msg_Chan(mrc:Timer)=
      sum(m:Msg, sum(b:Bool, msg_in_ch(m, b).Msg_Chan_W(on(2),m, b)))
250   +
      tick.Msg_Chan(off)<|not(expire(mrc))|>delta
      +
      tick.Msg_Chan(off)<|not(expire(mrc))|>delta

255 proc Msg_Chan_W(mrc:Timer, msg:Msg, b:Bool)=
      mdelay.Msg_Chan_D(off, msg,b)<|expire(mrc)|>delta
      +
      _mchbusy.Msg_Chan_W(mrc,msg, b)<|not(expire(mrc))|>delta
      +
260   tick.(Msg_Chan_W(pred(mrc),msg, b))<|not(expire(mrc))|>delta
      +
      tick.(Msg_Chan_W(pred(mrc),msg, b))<|not(expire(mrc))|>delta

    proc Msg_Chan_D(mrc:Timer, msg:Msg,  b:Bool)=
265   msg_out_ch(msg, b).Msg_Chan(off)
      +
      mlose.Msg_Chan(off)

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
270 %%% process receiver                                          %%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    proc Receiver_Wf(esn:Bool)=
       sum(m:Msg, sum(b:Bool, msg_in_rcv(m, b).Receiver_Wf1(m, b, esn)))
       +
275    tick.Receiver_Wf(esn)
       +
       tick.Receiver_Wf(esn)

    proc Receiver_Wf1(rm:Msg,  rsn:Bool, esn:Bool)=
280    vfvs.Receiver_Sa1( rsn, rm,esn)<|eq(rsn, esn)|>delta
       +
       vfws.Receiver_Sa(esn)<|not(eq(rsn, esn))|>delta

    proc Receiver_Sa1(rsn:Bool, rm:Msg, esn:Bool)=
285    _out(rm).Receiver_Sa(not(esn))

    proc Receiver_Sa(esn:Bool)=
       ack_out_rcv.Receiver_Wf(esn)

290 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%% process ack channel                                       %%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    proc Ack_Chan(arc:Timer)=
      ack_in_ch.Ack_Chan_W(on(2))
295   +
      tick.Ack_Chan(off)<|not(expire(arc))|>delta
      +
      tick.Ack_Chan(off)<|not(expire(arc))|>delta
    proc Ack_Chan_W(arc:Timer)=
300   adelay.Ack_Chan_D(off)<|expire(arc)|>delta
      +
      tick.(Ack_Chan_W(pred(arc)))<|not(expire(arc))|>delta
      +
      tick.(Ack_Chan_W(pred(arc)))<|not(expire(arc))|>delta
305 proc Ack_Chan_D(arc:Timer)=
      ack_out_ch.Ack_Chan(off)
```

```
  +
  alose.Ack_Chan(off)

310 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% process environment                                          %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
proc Env1(had1:Bool,had2:Bool) =
      in(msg1).Env1(T,F)<|and(eq(had1,F),eq(had2,F))|>delta
315  +in(msg2).Env1(T,T)<|and(eq(had1,T),eq(had2,F))|>delta
     +in(msg3).Env1(had1,had2)

proc Env2(had1:Bool,had2:Bool) =
      out(msg1).Env2(T,F)<|and(eq(had1,F),eq(had2,F))|>delta
320  +out(msg2).Env2(T,T)<|and(eq(had1,T),eq(had2,F))|>delta
     +out(msg3).Env2(had1,had2)

init(
 encap({_in, in, out, _out},
325 rename({_tick->tick},
 encap({tick, msg_out_ch, msg_in_rcv, ack_out_ch, ack_in_snd},
 rename({_tick->tick},encap({tick, msg_in_ch,msg_out_snd, mchbusy,_mchbusy}, Sender_Rh(
     known(on(0)),F) || Msg_Chan(off)))
 ||
 rename({_tick->tick},encap({tick, ack_in_ch, ack_out_rcv}, Receiver_Wf(F) || Ack_Chan(off)
     ))
330 )
 )
  || Env1(F,F) || Env2(F,F)
 )
)
```

C. Abstract Counterexample

```
__in(msg1)
msg_out(msg1,F)
tick
tick
5 mdelay
msg_in(msg1,F)
vfvs
__out(msg1)
ack_out
10 tick
tick
retransmit
adelay
msg_out(msg1,F)
15 ack_in
__in(msg2)
__mchbusy
tick
__mchbusy
20 tick
mdelay
msg_in(msg1,F)
vfws
msg_out(msg2,T)
25 ack_out
tick
tick
mdelay
adelay
30 ack_in
__in(msg3)
mlose
```

```
msg_out(msg3,F)
tick
tick
mdelay
msg_in(msg3,F)
vfws
ack_out
tick
tick
adelay
ack_in % Start of the cyclic base
__in(msg3)
msg_out(msg3,T)
tick
tick
mdelay
mlose
tick
tick
restransmit
msg_out(msg3,T)
tick
tick
mdelay
msg_in(msg3,T)
vfvs
__out(msg3)
ack_out
tick
tick
adelay
ack_in
__in(msg3)
msg_out(msg3,F)
tick
tick
mdelay
msg_in(msg3,F)
vfvs
__out(msg3)
ack_out
tick
tick
adelay
ack_in % End of cyclic base
```

D. Rule System for PAR specification

```
:-lib(ic).
:-lib(ic_symbolic).

% The constraints determined from mCRL spec and ATC.
t.
f:- fail.
bool(X):- X.
and(X,Y):- X,Y.
nand(X,Y):- (X,Y)->f;t.
or(X,Y):- X;Y.
nor(X,Y):- (X;Y)->f;t.

not(bool(t),bool(f)):-!.
not(bool(f),bool(t)):-!.

eq(bool(V0),bool(V1)) :- V0=V1.
eq(bool(t),bool(not bool(f))). % Let's try with these four...
```

```
eq(bool(not bool(f)),bool(t)).
eq(bool(f),bool(not bool(t))).
eq(bool(not bool(t)),bool(f)).

eq(msg(V0), msg(V1)) :-  V0=V1.
eq(nat(V1),nat(V2)):- V1#=V2.
eq(state(V0), state(V0)) :-  bool(t).
eq(state(one), state(x2p1(state(V0)))) :-  bool(f).
eq(state(one), state(x2p0(state(V0)))) :-  bool(f).
eq(state(x2p1(state(V0))), state(one)) :-  bool(f).
eq(state(x2p0(state(V0))), state(one)) :-  bool(f).
eq(state(x2p1(state(V0))), state(x2p0(state(V1)))) :-  bool(f).
eq(state(x2p0(state(V0))), state(x2p1(state(V1)))) :-  bool(f).
eq(state(x2p0(state(V0))), state(x2p0(state(V1)))) :-  bool(eq(state(V0), state(V1))).
eq(state(x2p1(state(V0))), state(x2p1(state(V1)))) :-  bool(eq(state(V0), state(V1))).
expire(timer(VInt0)) :-  VInt0=off, f.
expire(timer(VInt0)) :-  VInt0=on(nat(0)).
gt(nat(V1),nat(V2)):- V1#>V2.
intonat(msg(VInt0), nat(Result)) :-  VInt0=msg1, Result#=1.
intonat(msg(VInt0), nat(Result)) :-  VInt0=msg2, Result#=2.
intonat(msg(VInt0), nat(Result)) :-  VInt0=msg3, Result#=3.
neq(nat(V1),nat(V2)):- V1#\=V2.
neq(state(V0), state(V0)) :-  (bool(t)) -> bool(f) ; bool(t).
neq(state(one), state(x2p1(state(V0)))) :-  (bool(f)) -> bool(f) ; bool(t).
neq(state(one), state(x2p0(state(V0)))) :-  (bool(f)) -> bool(f) ; bool(t).
neq(state(x2p1(state(V0))), state(one)) :-  (bool(f)) -> bool(f) ; bool(t).
neq(state(x2p0(state(V0))), state(one)) :-  (bool(f)) -> bool(f) ; bool(t).
neq(state(x2p1(state(V0))), state(x2p0(state(V1)))) :-  (bool(f)) -> bool(f) ; bool(t).
neq(state(x2p0(state(V0))), state(x2p1(state(V1)))) :-  (bool(f)) -> bool(f) ; bool(t).
neq(state(x2p0(state(V0))), state(x2p0(state(V1)))) :-  (bool(eq(state(V0), state(V1)))) ->
     bool(f) ; bool(t).
neq(state(x2p1(state(V0))), state(x2p1(state(V1)))) :-  (bool(eq(state(V0), state(V1)))) ->
     bool(f) ; bool(t).
next(msg(VInt0), msg(Result)) :-  VInt0=msg1, Result=msg2.
next(msg(VInt0), msg(Result)) :-  VInt0=msg2, Result=msg3.
next(msg(VInt0), msg(Result)) :-  VInt0=msg3, Result=msg1.
ngt(nat(V1),nat(V2)):- V1#=<V2.
pred(nat(VInt0), nat(Result)) :-  VInt0#=1+V0, Result#=V0.
pred(nat(VInt0), nat(Result)) :-  VInt0#=0, Result#=0.
pred(timer(VInt0), timer(Result)) :-  VInt0=on(nat(V0)), Result=on(nat(VInt1)), pred(nat(V0
     ), nat(VInt1)).
pred(timer(VInt0), timer(Result)) :-  VInt0=off, Result=off.
succ(nat(VInt0), nat(Result)) :-  VInt0#=0, Result#=1.
succ(nat(V0), nat(Result)) :-  Result#=1+V0.

in(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(
    Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(
    Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(V0),timer(V1),timer(V2),msg(
    V3),bool(Vsn),state(Vs1),timer(Vmrc),msg(Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm)
    ,bool(Vesn),state(Vs3),timer(Varc),bool(V4),bool(V5),bool(Vhad10),bool(Vhad20)),local(
    msg(V6))) :-  V6=msg2, V5=t, V4=t, V3=msg2, V2=off, V1=off, V0=x2p1(state(one)), bool(
    and(bool(and(bool(eq(state(Vs0),state(x2p0(state(x2p0(state(one))))))),bool(expire(
    timer(Vsc))))),bool(and(bool(eq(bool(Vhad1),bool(t))),bool(eq(bool(Vhad2),bool(f))))))
    .
in(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(
    Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(
    Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(V0),timer(V1),timer(V2),msg(
    V3),bool(Vsn),state(Vs1),timer(Vmrc),msg(Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm)
    ,bool(Vesn),state(Vs3),timer(Varc),bool(V4),bool(V5),bool(Vhad10),bool(Vhad20)),local(
    msg(V6))) :-  V6=msg1, V5=f, V4=t, V3=msg1, V2=off, V1=off, V0=x2p1(state(one)), bool(
    and(bool(and(bool(eq(state(Vs0),state(x2p0(state(x2p0(state(one))))))),bool(expire(
    timer(Vsc))))),bool(and(bool(eq(bool(Vhad1),bool(f))),bool(eq(bool(Vhad2),bool(f))))))
    .
in(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(
```

```
    Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(
    Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(V0),timer(V1),timer(V2),msg(
    V3),bool(Vsn),state(Vs1),timer(Vmrc),msg(Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm)
    ,bool(Vesn),state(Vs3),timer(Varc),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),
    local(msg(V4))) :-  V4=msg3, V3=msg3, V2=off, V1=off, V0=x2p1(state(one)), bool(and(
    bool(and(bool(eq(state(Vs0),state(x2p0(state(x2p0(state(one))))))),bool(expire(timer(
    Vsc))))),bool(t))).
out(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(
    Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(
    Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(Vs0),timer(Vsc),timer(Vt),
    msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(Vmsg0),bool(Vb),state(V0),bool(V1),msg(
    V2),bool(V3),state(Vs3),timer(Varc),bool(Vhad1),bool(Vhad2),bool(V4),bool(V5)),local(
    msg(Vrm))) :-  V5=t, V4=t, not(bool(Vesn),bool(V3)), V2=msg3, V1=f, V0=one, bool(and(
    bool(eq(msg(Vrm),msg(msg2))),bool(and(bool(eq(state(Vs2),state(x2p0(state(one))))),bool
    (and(bool(eq(bool(Vhad10),bool(t))),bool(eq(bool(Vhad20),bool(f))))))))).
out(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(
    Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(
    Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(Vs0),timer(Vsc),timer(Vt),
    msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(Vmsg0),bool(Vb),state(V0),bool(V1),msg(
    V2),bool(V3),state(Vs3),timer(Varc),bool(Vhad1),bool(Vhad2),bool(V4),bool(V5)),local(
    msg(Vrm))) :-  V5=f, V4=t, not(bool(Vesn),bool(V3)), V2=msg3, V1=f, V0=one, bool(and(
    bool(eq(msg(Vrm),msg(msg1))),bool(and(bool(eq(state(Vs2),state(x2p0(state(one))))),bool
    (and(bool(eq(bool(Vhad10),bool(f))),bool(eq(bool(Vhad20),bool(f))))))))).
65 out(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(
    Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(
    Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(Vs0),timer(Vsc),timer(Vt),
    msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(Vmsg0),bool(Vb),state(V0),bool(V1),msg(
    V2),bool(V3),state(Vs3),timer(Varc),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),
    local(msg(Vrm))) :-  not(bool(Vesn),bool(V3)), V2=msg3, V1=f, V0=one, bool(and(bool(eq(
    msg(Vrm),msg(msg3))),bool(and(bool(eq(state(Vs2),state(x2p0(state(one))))),bool(t))))).
mchbusy(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),
    msg(Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),
    bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(V0),timer(V1),timer(V2)
    ,msg(Vmsg),bool(Vsn),state(V3),timer(Vmrc),msg(Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),
    msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(
    Vhad20)),local) :-  V3=x2p0(state(one)), V2=on(nat(VInt0)), VInt0#=1, V1=off, V0=one,
    bool(and(bool(or(bool(and(bool(eq(state(Vs0),state(one))),bool(expire(timer(Vt))))),
    bool(eq(state(Vs0),state(x2p1(state(one))))))),bool(and(bool(eq(state(Vs1),state(x2p0(
    state(one))))),bool(not(bool(expire(timer(Vmrc))))))))).
msg_out(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),
    msg(Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),
    bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(V0),timer(V1),timer(V2)
    ,msg(Vmsg),bool(Vsn),state(V3),timer(V4),msg(Vmsg),bool(Vsn),state(Vs2),bool(Vrsn),msg(
    Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20
    )),local(msg(Vmsg),bool(Vsn))) :-  V4=on(nat(VInt0)), VInt0#=2, V3=x2p0(state(one)), V2
    =off, V1=on(nat(VInt1)), VInt1#=4, V0=x2p0(state(one)), bool(and(bool(or(bool(and(bool(
    eq(state(Vs0),state(one))),bool(expire(timer(Vt))))),bool(eq(state(Vs0),state(x2p1(
    state(one))))))),bool(eq(state(Vs1),state(x2p1(state(one))))))).
mlose(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg
    (Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(
    Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(Vs0),timer(Vsc),timer(Vt),
    msg(Vmsg),bool(Vsn),state(V0),timer(V1),msg(V2),bool(V3),state(Vs2),bool(Vrsn),msg(Vrm)
    ,bool(Vesn),state(Vs3),timer(Varc),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),
    local) :-  V3=f, V2=msg3, V1=off, V0=x2p1(state(one)), bool(eq(state(Vs1),state(one))).
mdelay(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),
    msg(Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),
    bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(Vs0),timer(Vsc),timer(
    Vt),msg(Vmsg),bool(Vsn),state(V0),timer(V1),msg(Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),
    msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(
    Vhad20)),local) :-  V1=off, V0=one, bool(and(bool(eq(state(Vs1),state(x2p0(state(one)))
    )),bool(expire(timer(Vmrc))))).
70 retransmit(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc
    ),msg(Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),
    bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(V0),timer(V1),timer(V2)
```

```
    ,msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),
    msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(
    Vhad20)),local) :-  V2=off, V1=off, V0=x2p1(state(one)), bool(and(bool(eq(state(Vs0),
    state(x2p0(state(one))))),bool(expire(timer(Vsc))))).
udelay(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),
    msg(Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),
    bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(V0),timer(V1),timer(V2)
    ,msg(V3),bool(Vsn),state(Vs1),timer(Vmrc),msg(Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg
    (Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(
    Vhad20)),local) :-  V3=msg3, V2=off, V1=on(nat(VInt0)), VInt0#=1, V0=x2p0(state(x2p0(
    state(one)))), bool(and(bool(eq(state(Vs0),state(x2p0(state(x2p0(state(one))))))),bool(
    expire(timer(Vsc))))).
ack_out(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),
    msg(Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),
    bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(Vs0),timer(Vsc),timer(
    Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(Vmsg0),bool(Vb),state(V0),bool(V1),
    msg(V2),bool(Vesn),state(V3),timer(V4),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20
    )),local) :-  V4=on(nat(VInt0)), VInt0#=2, V3=x2p0(state(one)), V2=msg3, V1=f, V0=x2p0(
    state(x2p0(state(one)))), bool(and(bool(eq(state(Vs2),state(one))),bool(eq(state(Vs3),
    state(x2p1(state(one))))))).
adelay(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),
    msg(Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),
    bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(Vs0),timer(Vsc),timer(
    Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(Vmsg0),bool(Vb),state(Vs2),bool(Vrsn
    ),msg(Vrm),bool(Vesn),state(V0),timer(V1),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(
    Vhad20)),local) :-  V1=off, V0=one, bool(and(bool(eq(state(Vs3),state(x2p0(state(one)))
    )),bool(expire(timer(Varc))))).
alose(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg
    (Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(
    Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(Vs0),timer(Vsc),timer(Vt),
    msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),
    msg(Vrm),bool(Vesn),state(V0),timer(V1),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(
    Vhad20)),local) :-  V1=off, V0=x2p1(state(one)), bool(eq(state(Vs3),state(one))).
%vfvs(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg
    (Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(
    Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(Vs0),timer(Vsc),timer(Vt),
    msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(Vmsg0),bool(Vb),state(V0),bool(Vrsn),msg
    (Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(
    Vhad20)),local) :-  V0=x2p0(state(one)), bool(and(bool(eq(state(Vs2),state(x2p1(state(
    one))))),bool(eq(bool(Vrsn),bool(Vesn))))).
%vfws(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg
    (Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(
    Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(Vs0),timer(Vsc),timer(Vt),
    msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(Vmsg0),bool(Vb),state(V0),bool(V1),msg(
    V2),bool(Vesn),state(Vs3),timer(Varc),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)
    ),local) :-  V2=msg3, V1=f, V0=one, bool(and(bool(eq(state(Vs2),state(x2p1(state(one)))
    )),bool(neq(bool(Vrsn),bool(Vesn))))).
vfvs(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(
    Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(
    Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(Vs0),timer(Vsc),timer(Vt),
    msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(Vmsg0),bool(Vb),state(V0),bool(Vrsn),msg
    (Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(
    Vhad20)),local) :- V0=x2p0(state(one)), Vs2=x2p1(state(one)),Vrsn=Vesn.
vfws(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(
    Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(
    Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(Vs0),timer(Vsc),timer(Vt),
    msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(Vmsg0),bool(Vb),state(V0),bool(V1),msg(
    V2),bool(Vesn),state(Vs3),timer(Varc),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)
    ),local) :-  V2=msg3, V1=f, V0=one, Vs2=x2p1(state(one)),Vrsn\=Vesn.
tick(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(
    Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(
    Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(V0),timer(V1),timer(V2),msg(
    Vmsg),bool(Vsn),state(V3),timer(V4),msg(V5),bool(V6),state(V7),bool(V8),msg(V9),bool(
    Vesn),state(V10),timer(V11),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),local)
```

```
    :-  pred(timer(Varc), timer(V11)), V10=x2p0(state(one)), V9=msg3, V8=f, V7=x2p0(state(
    x2p0(state(one)))), V6=f, V5=msg3, V4=off, V3=x2p1(state(one)), V2=off, pred(timer(Vsc)
    , timer(V1)), V0=x2p0(state(one)), bool(and(bool(and(bool(and(bool(eq(state(Vs0),state(
    x2p0(state(one))))),bool(not(bool(expire(timer(Vsc))))))),bool(and(bool(eq(state(Vs1),
    state(x2p1(state(one))))),bool(not(bool(expire(timer(Vmrc))))))))),bool(and(bool(eq(
    state(Vs2),state(x2p0(state(x2p0(state(one))))))),bool(and(bool(eq(state(Vs3),state(
    x2p0(state(one))))),bool(not(bool(expire(timer(Varc))))))))))).
80 tick(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(
    Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(
    Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(V0),timer(V1),timer(V2),msg(
    Vmsg),bool(Vsn),state(V3),timer(V4),msg(V5),bool(V6),state(V7),bool(V8),msg(V9),bool(
    Vesn),state(V10),timer(V11),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),local)
    :-  V11=off, V10=x2p1(state(one)), V9=msg3, V8=f, V7=x2p0(state(x2p0(state(one)))), V6=
    f, V5=msg3, V4=off, V3=x2p1(state(one)), V2=off, pred(timer(Vsc), timer(V1)), V0=x2p0(
    state(one)), bool(and(bool(and(bool(and(bool(eq(state(Vs0),state(x2p0(state(one))))),
    bool(not(bool(expire(timer(Vsc))))))),bool(and(bool(eq(state(Vs1),state(x2p1(state(one)
    )))),bool(not(bool(expire(timer(Vmrc))))))))),bool(and(bool(eq(state(Vs2),state(x2p0(
    state(x2p0(state(one))))))),bool(and(bool(eq(state(Vs3),state(x2p1(state(one))))),bool(
    not(bool(expire(timer(Varc))))))))))).
   tick(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(
    Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(
    Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(V0),timer(V1),timer(V2),msg(
    Vmsg),bool(Vsn),state(V3),timer(V4),msg(Vmsg0),bool(Vb),state(V5),bool(V6),msg(V7),bool
    (Vesn),state(V8),timer(V9),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),local) :-
      pred(timer(Varc), timer(V9)), V8=x2p0(state(one)), V7=msg3, V6=f, V5=x2p0(state(x2p0(
    state(one)))), pred(timer(Vmrc), timer(V4)), V3=x2p0(state(one)), V2=off, pred(timer(
    Vsc), timer(V1)), V0=x2p0(state(one)), bool(and(bool(and(bool(and(bool(eq(state(Vs0),
    state(x2p0(state(one))))),bool(not(bool(expire(timer(Vsc))))))),bool(and(bool(eq(state(
    Vs1),state(x2p0(state(one))))),bool(not(bool(expire(timer(Vmrc))))))))),bool(and(bool(
    eq(state(Vs2),state(x2p0(state(x2p0(state(one))))))),bool(and(bool(eq(state(Vs3),state(
    x2p0(state(one))))),bool(not(bool(expire(timer(Varc))))))))))).
   tick(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(
    Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(
    Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(V0),timer(V1),timer(V2),msg(
    Vmsg),bool(Vsn),state(V3),timer(V4),msg(Vmsg0),bool(Vb),state(V5),bool(V6),msg(V7),bool
    (Vesn),state(V8),timer(V9),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),local) :-
      V9=off, V8=x2p1(state(one)), V7=msg3, V6=f, V5=x2p0(state(x2p0(state(one)))), pred(
    timer(Vmrc), timer(V4)), V3=x2p0(state(one)), V2=off, pred(timer(Vsc), timer(V1)), V0=
    x2p0(state(one)), bool(and(bool(and(bool(and(bool(eq(state(Vs0),state(x2p0(state(one)))
    )),bool(not(bool(expire(timer(Vsc))))))),bool(and(bool(eq(state(Vs1),state(x2p0(state(
    one))))),bool(not(bool(expire(timer(Vmrc))))))))),bool(and(bool(eq(state(Vs2),state(
    x2p0(state(x2p0(state(one))))))),bool(and(bool(eq(state(Vs3),state(x2p1(state(one))))),
    bool(not(bool(expire(timer(Varc))))))))))).
   tick(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(
    Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(
    Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(V0),timer(V1),timer(V2),msg(
    Vmsg),bool(Vsn),state(V3),timer(V4),msg(V5),bool(V6),state(V7),bool(V8),msg(V9),bool(
    Vesn),state(V10),timer(V11),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),local)
    :-  pred(timer(Varc), timer(V11)), V10=x2p0(state(one)), V9=msg3, V8=f, V7=x2p0(state(
    x2p0(state(one)))), V6=f, V5=msg3, V4=off, V3=x2p1(state(one)), pred(timer(Vt), timer(
    V2)), V1=off, V0=one, bool(and(bool(and(bool(and(bool(eq(state(Vs0),state(one))),bool(
    not(bool(expire(timer(Vt))))))),bool(and(bool(eq(state(Vs1),state(x2p1(state(one))))),
    bool(not(bool(expire(timer(Vmrc))))))))),bool(and(bool(eq(state(Vs2),state(x2p0(state(
    x2p0(state(one))))))),bool(and(bool(eq(state(Vs3),state(x2p0(state(one))))),bool(not(
    bool(expire(timer(Varc))))))))))).
   tick(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(
    Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(
    Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(V0),timer(V1),timer(V2),msg(
    Vmsg),bool(Vsn),state(V3),timer(V4),msg(V5),bool(V6),state(V7),bool(V8),msg(V9),bool(
    Vesn),state(V10),timer(V11),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),local)
    :-  V11=off, V10=x2p1(state(one)), V9=msg3, V8=f, V7=x2p0(state(x2p0(state(one)))), V6=
    f, V5=msg3, V4=off, V3=x2p1(state(one)), pred(timer(Vt), timer(V2)), V1=off, V0=one,
    bool(and(bool(and(bool(and(bool(eq(state(Vs0),state(one))),bool(not(bool(expire(timer(
    Vt))))))),bool(and(bool(eq(state(Vs1),state(x2p1(state(one))))),bool(not(bool(expire(
```

```
    timer(Vmrc))))))))),bool(and(bool(eq(state(Vs2),state(x2p0(state(x2p0(state(one))))))),
    bool(and(bool(eq(state(Vs3),state(x2p1(state(one))))),bool(not(bool(expire(timer(Varc))
    ))))))))).
85 tick(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(
    Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(
    Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(V0),timer(V1),timer(V2),msg(
    Vmsg),bool(Vsn),state(V3),timer(V4),msg(Vmsg0),bool(Vb),state(V5),bool(V6),msg(V7),bool
    (Vesn),state(V8),timer(V9),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),local) :-
      pred(timer(Varc), timer(V9)), V8=x2p0(state(one)), V7=msg3, V6=f, V5=x2p0(state(x2p0(
    state(one)))), pred(timer(Vmrc), timer(V4)), V3=x2p0(state(one)), pred(timer(Vt), timer
    (V2)), V1=off, V0=one, bool(and(bool(and(bool(and(bool(eq(state(Vs0),state(one))),bool(
    not(bool(expire(timer(Vt))))))),bool(and(bool(eq(state(Vs1),state(x2p0(state(one))))),
    bool(not(bool(expire(timer(Vmrc))))))))),bool(and(bool(eq(state(Vs2),state(x2p0(state(
    x2p0(state(one))))))),bool(and(bool(eq(state(Vs3),state(x2p0(state(one))))),bool(not(
    bool(expire(timer(Varc)))))))))))).
   tick(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(
    Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(
    Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(V0),timer(V1),timer(V2),msg(
    Vmsg),bool(Vsn),state(V3),timer(V4),msg(Vmsg0),bool(Vb),state(V5),bool(V6),msg(V7),bool
    (Vesn),state(V8),timer(V9),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),local) :-
      V9=off, V8=x2p1(state(one)), V7=msg3, V6=f, V5=x2p0(state(x2p0(state(one)))), pred(
    timer(Vmrc), timer(V4)), V3=x2p0(state(one)), pred(timer(Vt), timer(V2)), V1=off, V0=
    one, bool(and(bool(and(bool(and(bool(eq(state(Vs0),state(one))),bool(not(bool(expire(
    timer(Vt))))))),bool(and(bool(eq(state(Vs1),state(x2p0(state(one))))),bool(not(bool(
    expire(timer(Vmrc))))))))),bool(and(bool(eq(state(Vs2),state(x2p0(state(x2p0(state(one)
    )))))),bool(and(bool(eq(state(Vs3),state(x2p1(state(one))))),bool(not(bool(expire(timer
    (Varc)))))))))))).
   tick(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(
    Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(
    Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(V0),timer(V1),timer(V2),msg(
    V3),bool(Vsn),state(V4),timer(V5),msg(V6),bool(V7),state(V8),bool(V9),msg(V10),bool(
    Vesn),state(V11),timer(V12),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),local)
    :-  pred(timer(Varc), timer(V12)), V11=x2p0(state(one)), V10=msg3, V9=f, V8=x2p0(state(
    x2p0(state(one)))), V7=f, V6=msg3, V5=off, V4=x2p1(state(one)), V3=msg3, V2=off, pred(
    timer(Vsc), timer(V1)), V0=x2p0(state(x2p0(state(one)))), bool(and(bool(and(bool(and(
    bool(eq(state(Vs0),state(x2p0(state(x2p0(state(one))))))),bool(not(bool(expire(timer(
    Vsc))))))),bool(and(bool(eq(state(Vs1),state(x2p1(state(one))))),bool(not(bool(expire(
    timer(Vmrc))))))))),bool(and(bool(eq(state(Vs2),state(x2p0(state(x2p0(state(one))))))),
    bool(and(bool(eq(state(Vs3),state(x2p0(state(one))))),bool(not(bool(expire(timer(Varc))
    ))))))))).
   tick(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(
    Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(
    Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(V0),timer(V1),timer(V2),msg(
    V3),bool(Vsn),state(V4),timer(V5),msg(V6),bool(V7),state(V8),bool(V9),msg(V10),bool(
    Vesn),state(V11),timer(V12),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),local)
    :-  V12=off, V11=x2p1(state(one)), V10=msg3, V9=f, V8=x2p0(state(x2p0(state(one)))), V7
    =f, V6=msg3, V5=off, V4=x2p1(state(one)), V3=msg3, V2=off, pred(timer(Vsc), timer(V1)),
     V0=x2p0(state(x2p0(state(one)))), bool(and(bool(and(bool(and(bool(eq(state(Vs0),state(
    x2p0(state(x2p0(state(one))))))),bool(not(bool(expire(timer(Vsc))))))),bool(and(bool(eq
    (state(Vs1),state(x2p1(state(one))))),bool(not(bool(expire(timer(Vmrc))))))))),bool(and
    (bool(eq(state(Vs2),state(x2p0(state(x2p0(state(one))))))),bool(and(bool(eq(state(Vs3),
    state(x2p1(state(one))))),bool(not(bool(expire(timer(Varc)))))))))))).
   tick(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(
    Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(
    Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(V0),timer(V1),timer(V2),msg(
    V3),bool(Vsn),state(V4),timer(V5),msg(Vmsg0),bool(Vb),state(V6),bool(V7),msg(V8),bool(
    Vesn),state(V9),timer(V10),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),local) :-
      pred(timer(Varc), timer(V10)), V9=x2p0(state(one)), V8=msg3, V7=f, V6=x2p0(state(x2p0
    (state(one)))), pred(timer(Vmrc), timer(V5)), V4=x2p0(state(one)), V3=msg3, V2=off,
    pred(timer(Vsc), timer(V1)), V0=x2p0(state(x2p0(state(one)))), bool(and(bool(and(bool(
    and(bool(eq(state(Vs0),state(x2p0(state(x2p0(state(one))))))),bool(not(bool(expire(
    timer(Vsc))))))),bool(and(bool(eq(state(Vs1),state(x2p0(state(one))))),bool(not(bool(
    expire(timer(Vmrc))))))))),bool(and(bool(eq(state(Vs2),state(x2p0(state(x2p0(state(one)
    )))))),bool(and(bool(eq(state(Vs3),state(x2p0(state(one))))),bool(not(bool(expire(timer
```

```
(Varc))))))))))).
tick(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),msg(
    Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),bool(
    Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(V0),timer(V1),timer(V2),msg(
    V3),bool(Vsn),state(V4),timer(V5),msg(Vmsg0),bool(Vb),state(V6),bool(V7),msg(V8),bool(
    Vesn),state(V9),timer(V10),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),local) :-
      V10=off, V9=x2p1(state(one)), V8=msg3, V7=f, V6=x2p0(state(x2p0(state(one)))), pred(
    timer(Vmrc), timer(V5)), V4=x2p0(state(one)), V3=msg3, V2=off, pred(timer(Vsc), timer(
    V1)), V0=x2p0(state(x2p0(state(one)))), bool(and(bool(and(bool(and(bool(eq(state(Vs0),
    state(x2p0(state(x2p0(state(one))))))),bool(not(bool(expire(timer(Vsc))))))),bool(and(
    bool(eq(state(Vs1),state(x2p0(state(one))))),bool(not(bool(expire(timer(Vmrc))))))))),
    bool(and(bool(eq(state(Vs2),state(x2p0(state(x2p0(state(one))))))),bool(and(bool(eq(
    state(Vs3),state(x2p1(state(one))))),bool(not(bool(expire(timer(Varc))))))))))).
msg_in(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),
    msg(Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),
    bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(Vs0),timer(Vsc),timer(
    Vt),msg(Vmsg),bool(Vsn),state(V0),timer(V1),msg(V2),bool(V3),state(V4),bool(Vb),msg(
    Vmsg0),bool(Vesn),state(Vs3),timer(Varc),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(
    Vhad20)),local(msg(Vmsg0),bool(Vb))) :-  V4=x2p1(state(one)), V3=f, V2=msg3, V1=off, V0
    =x2p1(state(one)), bool(and(bool(eq(state(Vs1),state(one))),bool(eq(state(Vs2),state(
    x2p0(state(x2p0(state(one))))))))).
ack_in(global(state(Vs0),timer(Vsc),timer(Vt),msg(Vmsg),bool(Vsn),state(Vs1),timer(Vmrc),
    msg(Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(Vrm),bool(Vesn),state(Vs3),timer(Varc),
    bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),global(state(V0),timer(V1),timer(V2)
    ,msg(V3),bool(V4),state(Vs1),timer(Vmrc),msg(Vmsg0),bool(Vb),state(Vs2),bool(Vrsn),msg(
    Vrm),bool(Vesn),state(V5),timer(V6),bool(Vhad1),bool(Vhad2),bool(Vhad10),bool(Vhad20)),
    local) :-  V6=off, V5=x2p1(state(one)), not(bool(Vsn),bool(V4)), V3=msg3, V2=off, V1=on
    (nat(VInt0)), VInt0#=0, V0=x2p0(state(x2p0(state(one)))), bool(and(bool(eq(state(Vs0),
    state(x2p0(state(one))))),bool(eq(state(Vs3),state(one))))).
```

E. Rule System for the Violation Pattern

```
:- dynamic cyclic_state/1.

oracle:- s0(global(state(x2p0(state(x2p0(state(one))))),timer(on(nat(0))),timer(off),msg(
    msg3),bool(f),state(x2p1(state(one))),timer(off),msg(msg3),bool(f),state(x2p0(state(
    x2p0(state(one))))),
       bool(f),msg(msg3),bool(f),state(x2p1(state(one))),timer(off),bool(f),bool(f),bool(f)
           ,bool(f)),data,[["<init>",s0]]).
s0(Global,data,Trace):- in(Global,GlobalPrime,local(msg(msg1))),s1(GlobalPrime,data,[["in(
    msg1)",s1]|Trace]).

s1(Global,data,Trace):- out(Global,GlobalPrime,local(msg(msg1))),s2(GlobalPrime,data,[["out
    (msg1)",s2]|Trace]).
s1(Global,data,Trace):- msg_out(Global,GlobalPrime,local(msg(msg1),bool(f))), s1(
    GlobalPrime,data,[["msg_out(msg1,f)",s1]|Trace]).
s1(Global,data,Trace):- tick(Global,GlobalPrime,local), s1(GlobalPrime,data,[["tick",s1]|
    Trace]).
s1(Global,data,Trace):- mdelay(Global,GlobalPrime,local), s1(GlobalPrime,data,[["mdelay",s1
    ]|Trace]).
s1(Global,data,Trace):- msg_in(Global,GlobalPrime,local(msg(msg1),bool(f))), s1(GlobalPrime
    ,data,[["msg_in(msg1,f)",s1]|Trace]).
s1(Global,data,Trace):- vfvs(Global,GlobalPrime,local), s1(GlobalPrime,data,[["vfvs",s1]|
    Trace]).

s2(Global,data,Trace):- in(Global,GlobalPrime,local(msg(msg2))),s3(GlobalPrime,data,[["in(
    msg2)",s3]|Trace]).
s2(Global,data,Trace):- ack_out(Global,GlobalPrime,local), s2(GlobalPrime,data,[["ack_out",
    s2]|Trace]).
s2(Global,data,Trace):- tick(Global,GlobalPrime,local), s2(GlobalPrime,data,[["tick",s2]|
    Trace]).
s2(Global,data,Trace):- adelay(Global,GlobalPrime,local), s2(GlobalPrime,data,[["adelay",s2
    ]|Trace]).
s2(Global,data,Trace):- retransmit(Global,GlobalPrime,local), s2(GlobalPrime,data,[["
    retransmit",s2]|Trace]).
```

```
s2(Global,data,Trace):- msg_out(Global,GlobalPrime,local(msg(msg1),bool(f))), s2(
    GlobalPrime,data,[["msg_out(msg1,f)",s2]|Trace]).
20 s2(Global,data,Trace):- ack_in(Global,GlobalPrime,local), s2(GlobalPrime,data,[["ack_in",s2
    ]|Trace]).

s3(Global,data,Trace):- in(Global,GlobalPrime,local(msg(msg3))),s4(GlobalPrime,data,[["in(
    msg3)",s4]|Trace]).
s3(Global,data,Trace):- mchbusy(Global,GlobalPrime,local),s3(GlobalPrime,data,[["__mchbusy"
    ,s3]|Trace]).
s3(Global,data,Trace):- tick(Global,GlobalPrime,local),s3(GlobalPrime,data,[["tick",s3]|
    Trace]).
25 s3(Global,data,Trace):- mdelay(Global,GlobalPrime,local),s3(GlobalPrime,data,[["mdelay",s3
    ]|Trace]).
s3(Global,data,Trace):- msg_in(Global,GlobalPrime,local(msg(msg1),bool(f))),s3(GlobalPrime,
    data,[["msg_in(msg1,f)",s3]|Trace]).
s3(Global,data,Trace):- msg_out(Global,GlobalPrime,local(msg(msg2),bool(t))),s3(GlobalPrime
    ,data,[["msg_out(msg2,t)",s3]|Trace]).
s3(Global,data,Trace):- vfws(Global,GlobalPrime,local),s3(GlobalPrime,data,[["vfws",s3]|
    Trace]).
s3(Global,data,Trace):- ack_out(Global,GlobalPrime,local),s3(GlobalPrime,data,[["ack_out",
    s3]|Trace]).
30 s3(Global,data,Trace):- adelay(Global,GlobalPrime,local),s3(GlobalPrime,data,[["adelay",s3
    ]|Trace]).
s3(Global,data,Trace):- ack_in(Global,GlobalPrime,local),s3(GlobalPrime,data,[["ack_in",s3
    ]|Trace]).

s4(Global,data,Trace):- ack_in(Global,GlobalPrime,local), s5(GlobalPrime,data,[["ack_in",s5
    ]|Trace]).
s4(Global,data,Trace):- mlose(Global,GlobalPrime,local), s4(GlobalPrime,data,[["mlose",s4]|
    Trace]).
35 s4(Global,data,Trace):- msg_out(Global,GlobalPrime,local(msg(msg3),bool(f))), s4(
    GlobalPrime,data,[["msg_out(msg3,f)",s4]|Trace]).
s4(Global,data,Trace):- tick(Global,GlobalPrime,local), s4(GlobalPrime,data,[["tick",s4]|
    Trace]).
s4(Global,data,Trace):- mdelay(Global,GlobalPrime,local), s4(GlobalPrime,data,[["mdelay",s4
    ]|Trace]).
s4(Global,data,Trace):- msg_in(Global,GlobalPrime,local(msg(msg3),bool(f))), s4(GlobalPrime
    ,data,[["msg_in(msg3,f)",s4]|Trace]).
s4(Global,data,Trace):- vfws(Global,GlobalPrime,local), s4(GlobalPrime,data,[["vfws",s4]|
    Trace]).
40 s4(Global,data,Trace):- ack_out(Global,GlobalPrime,local), s4(GlobalPrime,data,[["ack_out",
    s4]|Trace]).
s4(Global,data,Trace):- adelay(Global,GlobalPrime,local), s4(GlobalPrime,data,[["adelay",s4
    ]|Trace]).

s5(Global,data,Trace):- write(Global),nl,write("----------"),nl,assert(cyclic_state(Global)
    ),in(Global,GlobalPrime,local(msg(msg3))),s6(GlobalPrime,data,[["in(msg3)",s6]|Trace]).

45 s6(Global,data,Trace):- out(Global,GlobalPrime,local(msg(msg3))),s7(GlobalPrime,data,[["out
    (msg3)",s7]|Trace]).
s6(Global,data,Trace):- msg_out(Global,GlobalPrime,local(msg(msg3),bool(t))), s6(
    GlobalPrime,data,[["msg_out(msg3,t)",s6]|Trace]).
s6(Global,data,Trace):- tick(Global,GlobalPrime,local), s6(GlobalPrime,data,[["tick",s6]|
    Trace]).
s6(Global,data,Trace):- mdelay(Global,GlobalPrime,local), s6(GlobalPrime,data,[["mdelay",s6
    ]|Trace]).
s6(Global,data,Trace):- msg_in(Global,GlobalPrime,local(msg(msg3),bool(t))), s6(GlobalPrime
    ,data,[["msg_in(msg3,t)",s6]|Trace]).
50 s6(Global,data,Trace):- vfvs(Global,GlobalPrime,local), s6(GlobalPrime,data,[["vfvs",s6]|
    Trace]).

s7(Global,data,Trace):- in(Global,GlobalPrime,local(msg(msg3))),s8(GlobalPrime,data,[["in(
    msg3)",s8]|Trace]).
s7(Global,data,Trace):- ack_out(Global,GlobalPrime,local), s7(GlobalPrime,data,[["ack_out",
```

```
    s7]|Trace]).
s7(Global,data,Trace):- tick(Global,GlobalPrime,local), s7(GlobalPrime,data,[["tick",s7]|
    Trace]).
s7(Global,data,Trace):- adelay(Global,GlobalPrime,local), s7(GlobalPrime,data,[["adelay",s7
    ]|Trace]).
s7(Global,data,Trace):- ack_in(Global,GlobalPrime,local), s7(GlobalPrime,data,[["ack_in",s7
    ]|Trace]).

s8(Global,data,Trace):- out(Global,GlobalPrime,local(msg(msg3))),s9(GlobalPrime,data,[["out
    (msg3)",s9]|Trace]).
s8(Global,data,Trace):- msg_out(Global,GlobalPrime,local(msg(msg3),bool(f))), s8(
    GlobalPrime,data,[["msg_out(msg3,f)",s8]|Trace]).
s8(Global,data,Trace):- tick(Global,GlobalPrime,local), s8(GlobalPrime,data,[["tick",s8]|
    Trace]).
s8(Global,data,Trace):- mdelay(Global,GlobalPrime,local), s8(GlobalPrime,data,[["mdelay",s8
    ]|Trace]).
s8(Global,data,Trace):- msg_in(Global,GlobalPrime,local(msg(msg3),bool(f))), s8(GlobalPrime
    ,data,[["msg_in(msg3,f)",s8]|Trace]).
s8(Global,data,Trace):- vfvs(Global,GlobalPrime,local), s8(GlobalPrime,data,[["vfvs",s8]|
    Trace]).

s9(Global,data,Trace):- ack_in(Global,GlobalPrime,local), s10(GlobalPrime,data,[["ack_in",
    s10]|Trace]).
%s9(Global,data,Trace):- in(Global,GlobalPrime,local(msg(msg3))),s10(GlobalPrime,data,[["in
    (msg3)",s10]|Trace]).
s9(Global,data,Trace):- ack_out(Global,GlobalPrime,local), s9(GlobalPrime,data,[["ack_out",
    s9]|Trace]).
s9(Global,data,Trace):- tick(Global,GlobalPrime,local), s9(GlobalPrime,data,[["tick",s9]|
    Trace]).
s9(Global,data,Trace):- adelay(Global,GlobalPrime,local), s9(GlobalPrime,data,[["adelay",s9
    ]|Trace]).

s10(Global,data,Trace):- cyclic_state(Global),write(Global),nl,write("----------"),nl,
    printtrace(Trace).

printtrace([]) :- !.
printtrace([A]) :- printcombo(A),!.
printtrace([A|B]) :- printtrace(B),!,printcombo(A).

printcombo(["<init>",S]) :- write(S),!.
printcombo([L,S]) :- write("␣->("), write(L), write(")␣"), write(S), nl, write(S).
```

F. Prolog Solution: Concrete Counterexample

```
Output Prolog:
==============
[eclipse 3]: oracle.
global(state(x2p0(state(x2p0(state(one))))), timer(on(nat(0))), timer(off), msg(msg3), bool
    (t), state(x2p1(state(one))), timer(off), msg(msg3), bool(f), state(x2p0(state(x2p0(
    state(one))))), bool(f), msg(msg3), bool(t), state(x2p1(state(one))), timer(off), bool(
    t), bool(t), bool(t), bool(f))
----------
global(state(x2p0(state(x2p0(state(one))))), timer(on(nat(0))), timer(off), msg(msg3), bool
    (t), state(x2p1(state(one))), timer(off), msg(msg3), bool(f), state(x2p0(state(x2p0(
    state(one))))), bool(f), msg(msg3), bool(t), state(x2p1(state(one))), timer(off), bool(
    t), bool(t), bool(t), bool(f))
----------
s0 ->(in(msg1)) s1
s1 ->(msg_out(msg1,f)) s1
s1 ->(tick) s1
s1 ->(tick) s1
s1 ->(mdelay) s1
s1 ->(msg_in(msg1,f)) s1
s1 ->(vfvs) s1
s1 ->(out(msg1)) s2
```

```
s2 ->(ack_out) s2
s2 ->(tick) s2
s2 ->(tick) s2
s2 ->(adelay) s2
20 s2 ->(retransmit) s2
s2 ->(msg_out(msg1,f)) s2
s2 ->(ack_in) s2
s2 ->(in(msg2)) s3
s3 ->(__mchbusy) s3
25 s3 ->(tick) s3
s3 ->(__mchbusy) s3
s3 ->(tick) s3
s3 ->(mdelay) s3
s3 ->(msg_in(msg1,f)) s3
30 s3 ->(msg_out(msg2,t)) s3
s3 ->(vfws) s3
s3 ->(ack_out) s3
s3 ->(tick) s3
s3 ->(tick) s3
35 s3 ->(mdelay) s3
s3 ->(adelay) s3
s3 ->(ack_in) s3
s3 ->(in(msg3)) s4
s4 ->(mlose) s4
40 s4 ->(msg_out(msg3,f)) s4
s4 ->(tick) s4
s4 ->(tick) s4
s4 ->(mdelay) s4
s4 ->(msg_in(msg3,f)) s4
45 s4 ->(vfws) s4
s4 ->(ack_out) s4
s4 ->(tick) s4
s4 ->(tick) s4
s4 ->(adelay) s4
50 s4 ->(ack_in) s5
s5 ->(in(msg3)) s6
s6 ->(msg_out(msg3,t)) s6
s6 ->(tick) s6
s6 ->(tick) s6
55 s6 ->(mdelay) s6
s6 ->(msg_in(msg3,t)) s6
s6 ->(vfvs) s6
s6 ->(out(msg3)) s7
s7 ->(ack_out) s7
60 s7 ->(tick) s7
s7 ->(tick) s7
s7 ->(adelay) s7
s7 ->(ack_in) s7
s7 ->(in(msg3)) s8
65 s8 ->(msg_out(msg3,f)) s8
s8 ->(tick) s8
s8 ->(tick) s8
s8 ->(mdelay) s8
s8 ->(msg_in(msg3,f)) s8
70 s8 ->(vfvs) s8
s8 ->(out(msg3)) s9
s9 ->(ack_out) s9
s9 ->(tick) s9
s9 ->(tick) s9
75 s9 ->(adelay) s9
s9 ->(ack_in) s10
s10
Yes (0.16s cpu, solution 1, maybe more) ? _
```