Synchronous Closing and Flow Analysis for Model Checking Timed Systems

Natalia Ioustinova[1], Natalia Sidorova[2], and Martin Steffen[3]

[1] Department of Software Engineering, CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Natalia.Ioustinova@cwi.nl
[2] Department of Mathematics and Computer Science
Eindhoven University of Technology
Den Dolech 2, P.O. Box 513,
5612 MB Eindhoven, The Netherlands
n.sidorova@tue.nl
[3] Institute of Computer Science and Applied Mathematics
Christian-Albrechts-Universität
Hermann-Rodewaldstr. 3,
24118 Kiel, Germany
ms@informatik.uni-kiel.de

Abstract. Formal methods, in particular model checking, are increasingly accepted as integral part of system development. With large software systems beyond the range of fully automatic verification, however, a combination of decomposition and abstraction techniques is needed. To model check components of a system, a standard approach is to close the component with an abstraction of its environment, as standard model checkers often do not handle open reactive systems directly. To make it useful in practice, the closing of the component should be automatic, both for data and for control abstraction. Specifically for model checking asynchronous open systems, external input queues should be removed, as they are a potential source of a combinatorial state explosion.
In this paper we investigate a class of environmental processes for which the *asynchronous* communication scheme can safely be replaced by a *synchronous* one. Such a replacement is possible only if the environment is constructed under rather a severe restriction on the behavior, which can be partially softened via the use of a discrete-time semantics. We employ *data-flow analysis* to detect instances of variables and timers influenced by the data passing between the system and the environment.

Keywords: formal methods, software model checking, abstraction, flow analysis, asynchronous communication, open components, program transformation

1 Introduction

Model checking [8] is well-accepted for the verification of reactive systems. To alleviate the notorious state-space explosion problem, a host of techniques has been invented, including partial-order reduction [12,32] and abstraction [23,8,10].

As standard model checkers, e.g., Spin [16], cannot handle open systems, one has to construct a closed model, and a problem of practical importance is how to *close* open systems. This is commonly done by adding an environment process that must exhibit at least all the behavior of the real environment. In the framework of the assume-guarantee paradigm, the environment should model the behavior corresponding to the verified properties of the components forming the environment. However, the way of closing should be well-considered to counter the state-space explosion problem. This is especially true in the context of model checking programs with an *asynchronous* message-passing communication model —- sending arbitrary message streams to the unbounded input queues would immediately lead to an infinite state space, unless some assumptions restricting the environment behavior are incorporated in the closing process. Even so, adding an environment process may result in a combinatorial explosion caused by all combinations of messages in the input queues.

A desirable solution would be to construct an environment that communicates to the system *synchronously.* In [29] such an approach is considered for the simplest safe abstraction of the environment, the *chaotically* behaving environment: the outside chaos is *embedded* into the system's processes, which corresponds to the synchronous communication scheme. Though useful at a first verification phase, the chaotic environment may be too general. Here, we investigate for what kind of processes, apart from the chaotic one, the asynchronous communication can be safely replaced with the synchronous one. To make such a replacement possible, the system should be not reactive — it should either only send or only receive messages. However, when we restrict our attention to systems with the discrete-time semantics like the ones of [15,3], this requirement can be softened in that the restrictions are imposed on time slices instead of whole runs: In every time slice, the environmental process can either only receive messages, or it can both send and receive messages under condition that inputs do not change the state of the environment process.

Another problem the closing must address is that the *data* carried with the messages are usually drawn from some infinite data domains. For *data abstraction,* we combine the approaches from [29] and [17]. The main idea is to condense data exchanged with the environment into a single abstract value $\mathbb{T}$ to deal with the infinity of environmental data. We employ *data-flow analysis* to detect instances of chaotically influenced variables and timers and remove them. Based on the result of the data flow analysis, the system S is transformed into a *closed* system $S^\sharp$ which shows more behavior in terms of traces than the original one. For formulas of next-free LTL [26,22], we thus get the desired property preservation: if $S^\sharp \models \varphi$ then $S \models \varphi$.

The main target application are protocols specified in SDL (*Specification and Description Language*) [28]. As verification tool, we use the well-known Spin model checker. Our method is implemented as transformations of Promela-programs, Spin's input language. With this tool we show experiments on a real-life protocol to estimate the effect of removing queues on the state space.

The rest of the paper is organized as follows. In Section 2 we fix syntax and semantics of the language. In Section 3 we describe under which condition the asynchronous communication with the environment can be replaced by synchronous one. In Section 4 we abstract from the data exchanged with the environment and give a data-flow algorithm to optimize the system for model checking. In Section 5 we show some experimental results and in Section 6 we discuss related and future work.

2 Semantics

In this section we fix syntax and operational semantics we work with. Our model is based on asynchronously communicating state machines with top-level concurrency. The communication is done via *channels* and we assume a fixed set $Chan$ of channel names for each program, with $c, c', \ldots$ as typical elements. The set of channel names is partitioned into input channels $Chan_i$ and output channels $Chan_o$, and we write $c_i, c'_o, \ldots$ to denote membership of a channel to one of these classes. A program $Prog$ is given as the parallel composition $\Pi_{i=1}^{n} P_i$ of a finite number of processes.

A process P is described by a tuple $(in, out, Var, Loc, Edg, \sigma_{init})$, where in and out are finite sets of *input* resp. *output* channel names of the process, Var denotes a finite set of variables, Loc denotes a finite set of *locations* or control states, and σ_{init} is the initial state. We assume the sets of variables Var_i of processes P_i in a program $Prog = \Pi_{i=1}^{n} P_i$ to be disjoint. An *edge* of the state machine describes a change of state by performing an *action* from a set Act; the set $Edg \subseteq Loc \times Act \times Loc$ denotes the set of edges. For an edge $(l, \alpha, \hat{l}) \in Edg$ of P, we write more suggestively $l \longrightarrow_{\alpha} \hat{l}$.

A mapping from variables to values is called a valuation; we denote the set of valuations by $Val = Var \rightarrow D$. We assume standard data domains such as $\mathbb{N}$, $Bool$, etc., where we write D when leaving the data domain unspecified, and we silently assume all expressions to be well-typed. A location together with a valuation of process variables define a *state* of a process. The set of process states is defined as $\Sigma = Loc \times Val$, and each process has one designated initial state $\sigma_{init} = (l_{init}, \eta_{init})$.

Processes communicate by exchanging *signals* (carrying values) over channels. Signals coming from the environment form the set of external signals Sig_{ext}. Signals that participate in the communication within the system belong to the set Sig_{int} of internal signals. Note that both signal sets are not necessarily disjoint.

As untimed actions, we distinguish (1) *input* over a channel c of a signal s containing a value to be assigned to a local variable, (2) *sending* over a channel c a signal s together with a value described by an expression, and (3) *assignments*. We assume the inputs to be unguarded, while output and assignment are *guarded* by a boolean expression g, its guard. The three classes of actions are written as $c?s(x)$, $g \triangleright c!s(e)$, and $g \triangleright x := e$, respectively, and we use α, $\alpha' \ldots$ when leaving an action unspecified.

$$\frac{l \longrightarrow_{c?s(x)} \hat{l} \in Edg}{(l,\eta) \rightarrow_{c_i?(s,v)} (\hat{l}, \eta_{[x \mapsto v]})} \textsc{Input} \qquad \frac{l \longrightarrow_{c?s(x)} \hat{l} \in Edg \Rightarrow s' \neq s}{(l,\eta) \rightarrow_{c_i?(s',v)} (l,\eta)} \textsc{Discard}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, c!(s,e)} \hat{l} \in Edg \qquad [\![g]\!]_\eta = true \qquad [\![e]\!]_\eta = v}{(l,\eta) \rightarrow_{c_o!(s,v)} (\hat{l}, \eta)} \textsc{Output}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, x:=e} \hat{l} \in Edg \qquad [\![g]\!]_\eta = true \qquad [\![e]\!]_\eta = v}{(l,\eta) \rightarrow_{\tau} (\hat{l}, \eta_{[x \mapsto v]})} \textsc{Assign}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, set\ t:=e} \hat{l} \in Edg \qquad [\![g]\!]_\eta = true \qquad [\![e]\!]_\eta = v}{(l,\eta) \rightarrow_{\tau} (\hat{l}, \eta_{[t \mapsto on(v)]})} \textsc{Set}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, reset\ t} \hat{l} \in Edg \qquad [\![g]\!]_\eta = true}{(l,\eta) \rightarrow_{\tau} (\hat{l}, \eta_{[t \mapsto off]})} \textsc{Reset} \qquad \frac{blocked(\sigma)}{\sigma \rightarrow_{tick} \sigma_{[t \mapsto (t-1)]}} \textsc{Tick}_P$$

$$\frac{l \longrightarrow_{g_t \,\triangleright\, reset\ t} \hat{l} \in Edg \qquad [\![t]\!]_\eta = on(0)}{(l,\eta) \rightarrow_{\tau} (\hat{l}, \eta_{[t \mapsto off]})} \textsc{Timeout}$$

$$\frac{(l \longrightarrow_{\alpha} \hat{l} \in Edg \Rightarrow \alpha \neq g_t \triangleright reset\ t) \qquad [\![t]\!]_\eta = on(0)}{(l,\eta) \rightarrow_{\tau} (l, \eta_{[t \mapsto off]})} \textsc{TDiscard}$$

Table 1. Step semantics for one process

Time aspects of a system behavior are specified by actions dealing with *timers*. Each process has a finite set of timer variables (with typical elements $t, t'_1, \ldots$). A timer can be either *set* to a value, i.e., activated to run for the designated period, or *reset*, i.e., deactivated, which corresponds to timer values $on(n)$, off, respectively. Setting and resetting are expressed by guarded actions of the form $g \triangleright set\ t := e$ and $g \triangleright reset\ t$. If a timer expires, i.e., the value of a timer becomes zero, it can cause a *timeout,* upon which the timer is reset. The timeout action is denoted by $g_t \triangleright reset\ t$, where the timer guard g_t expresses the fact that the action can only be taken upon expiration.

The behavior of a single process is then given by sequences of states $\sigma_{init} = \sigma_0 \rightarrow_\lambda \sigma_1 \rightarrow_\lambda \ldots$ starting from the initial one. The step semantics is given as a labelled transition relation $\rightarrow_\lambda \subseteq \Sigma \times Lab \times \Sigma$ between states. The set of labels Lab is formed by τ-labels for internal steps, $tick$-labels for time progression and communication labels. Communication label, either input or output are of the form $c?(s, v)$ resp. $c!(s, v)$. Depending on the location, the valuation, and the potential next actions, the possible successor states are given by the rules of Table 1.

Inputting a signal with a value via a channel means reading a value belonging to a matching signal from the channel and updating the local valuation accordingly (cf. rule Input), where $\eta_{[x \mapsto v]}$ stands for the valuation equaling η for all $y \in Var$ except for $x \in Var$, where $\eta_{[x \mapsto v]}(x) = v$ holds instead. A specific feature commonly used for communicating finite state machines (e.g. in SDL-

92 [27]) is captured by rule DISCARD: If the input value cannot be reacted upon at the current control state, i.e., if there is no input action originating from the location treating this signal, then the message is just discarded, leaving control state and valuation unchanged. The automaton is therefore input-enabled: it cannot refuse to accept a message; it may throw it away, though.

Unlike inputs, outputs are guarded, so sending a message involves evaluating the guard and the expression according to the current valuation (cf. rule OUTPUT). Assignment in ASSIGN works analogously, except that the step is internal. We assume for the non-timer guards, that at least one of them evaluates to true in each state. At the SDL source language, this assumption corresponds to the natural requirement that each conditional construct must cover all cases, for instance by having at least a default branch. The system should not block because of a non-covered alternative in a decision-construct [25].

Concerning the temporal behavior, timers are treated in valuations as variables, distinguishing active and deactivated timer. We use *off* to represent inactive timers. The value of an active timer shows the delay left until timer expiration. The *set*-command activates a timer, setting its value to the specified period until timer expiration, and *reset* deactivates the timer. Both actions are guarded (cf. rules SET and RESET). A timeout may occur, if an active timer has expired, i.e., reached zero (cf. rule TIMEOUT).

Time elapses by counting down active timers till zero, which happens in case no untimed actions are possible. In rule TICK$_P$, this is expressed by the predicate *blocked* on states: $blocked(\sigma)$ holds if no move is possible except either a *tick*-step or a reception of a message, i.e., if $\sigma \rightarrow_\lambda$ for some label λ, then $\lambda = tick$ or $\lambda = c?(s, v)$. In other words, the time-elapsing steps are those with *least priority.* The counting down of the timers is written $\eta_{[t \mapsto (t-1)]}$, by which we mean, all currently active timers are decreased by one, i.e., $on(n + 1) - 1 = on(n)$, non-active timers are not affected. Note that the operation is undefined for $on(0)$, which is justified later by Lemma 1.

In SDL, timeouts are often considered as specific timeout *messages* kept in a queue like any other message, and timer-expiration consequently is seen as adding a timeout-message to the queue. We use an equivalent presentation of this semantics, where timeouts are not put into the input queue, but are modelled more directly by guards. The equivalence of timeouts-by-guards and timeouts-as-messages in the presence of SDL's asynchronous communication model is argued for in [3]. The time semantics for SDL chosen here is not the only one conceivable (see e.g. [6] for a broader discussion of the use of timers in SDL). The semantics we use is the one described in [15,3], and is also implemented in DTSPIN [2,11], a discrete time extension of the SPIN model checker.

In the asynchronous communication model, a process receives messages via channels modelled as queues. We write ϵ for the empty queue; $(s, v) :: q$ denotes a queue with message (s, v) at the head of the queue, i.e., (s, v) is the message to be input next; likewise the queue $q :: (s, v)$ contains (s, v) most recently entered; Q denotes the set of possible queues. We model the queues implementing asynchronous channels explicitly as separate entities of the form (c, q), consisting of the channel name together with its queue content. We sometimes refer to the channel process (c, q) just by its name c. We require for the input and the output channel names of channel c to be $in(c) = c_o$ and $out(c) = c_i$ resp. The operational rules for queues are shown in Table 2.

$$\frac{}{(c, q) \rightarrow_{c_o?(s,v)} (c, q :: (s, v))}\ \textsc{In} \qquad \frac{blocked(c, q)}{(c, q) \rightarrow_{tick} (c, q)}\ \textsc{Tick}_Q$$

$$\frac{}{(c, (s, v) :: q) \rightarrow_{c_i!(s,v)} (c, q)}\ \textsc{Out}$$

Table 2. Step semantics for a channel c

In analogy to the *tick*-steps for processes, a queue can perform a *tick*-step iff the only steps possible are input or tick-steps, as captured again by the *blocked*-predicate (cf. rule Tick). Note that a queue is blocked and can therefore tick only if it is empty, and that a queue does not contain any timers. Hence, the counting down operation $[t \mapsto (t-1)]$ has no effect and is therefore omitted in the rule $\textsc{Tick}_Q$ of Table 2.

$$\frac{\sigma_i \rightarrow_{c!(s,v)} \hat{\sigma}_i \qquad \sigma_j \rightarrow_{c?(s,v)} \hat{\sigma}_j \qquad i \neq j}{(\ldots, \sigma_i, \ldots, \sigma_j, \ldots) \rightarrow_\tau (\ldots, \hat{\sigma}_i, \ldots, \hat{\sigma}_j, \ldots)}\ \textsc{Comm}$$

$$\frac{\sigma_1 \rightarrow_{tick} \hat{\sigma}'_1 \ldots \sigma_n \rightarrow_{tick} \hat{\sigma}'_n}{(\sigma_1, \ldots, \sigma_n) \rightarrow_{tick} (\hat{\sigma}_1, \ldots, \hat{\sigma}_n)}\ \textsc{Tick}$$

$$\frac{\sigma_i \rightarrow_{c?(s,v)} \hat{\sigma}_i \qquad c \in ext(S)}{(\ldots, \sigma_i, \ldots) \rightarrow_{c?(s,v)} (\ldots, \hat{\sigma}_i, \ldots)}\ \textsc{Interleave}_{in}$$

$$\frac{\sigma_i \rightarrow_{c!(s,v)} \hat{\sigma}_i \qquad c \in ext(S)}{(\ldots, \sigma_i, \ldots) \rightarrow_{c!(s,v)} (\ldots, \hat{\sigma}_i, \ldots)}\ \textsc{Interleave}_{out}$$

$$\frac{\sigma_i \rightarrow_\tau \hat{\sigma}_i}{(\ldots, \sigma_i, \ldots) \rightarrow_\tau (\ldots, \hat{\sigma}_i, \ldots)}\ \textsc{Interleave}_\tau$$

Table 3. Parallel composition

A global semantics of a system S is given by a parallel composition of labelled transition systems modelling processes and channels of the specification. The semantics of the parallel composition $S = S_1 \parallel \ldots \parallel S_n$ is given by the rules of Table 3, where $ext(S)$ is used to denote the set of external channel names. Since we assumed the variable sets of the components to be disjoint, the combined state is defined as the product. We write $[\![x]\!]_\sigma$ for the value $[\![x]\!]_{\sigma_i}$ of x, for one state σ_i being part of σ; analogously, we use the notation $[\![e]\!]_\sigma$ for the value of e in σ. The *initial* state of a parallel composition is given by the array of initial process states together with (c, ϵ) for channels in *Chan*. We call a sequence $\sigma_{init} = \sigma_0 \rightarrow_\lambda \sigma_1 \rightarrow_\lambda \ldots$ starting from an initial state a *run*.

Communication between two processes is done by exchanging a common signal and a value over a channel. According to the syntactic restriction on the use of communication labels, only synchronisation between a process and a channel may happen. Sending of a signal over the channel means synchronising an output step of the process with an input step of the queue, i.e. a $c_o!(s, v)$ step of the process is synchronised with a $c_o?(s, v)$ step of the channel c. Receiving is accomplished by synchronising an output step, which removes first element from the channel queue, with an input step of the process. As defined by the rule Comm of Table 3, systems perform common steps synchronously. The result of communication is relabelled to τ.

Communication steps of two partners may synchronize, if they use the same channel name. Communication steps may be interleaved as in rules Interleave$_{in}$ and Interleave$_{out}$ provided the channel name belongs to the set of external channel names $ext(S)$ of the system. As far as τ steps are concerned, each system can act on its own according to rule Interleave$_\tau$.

Lemma 1. *Let S be a system and $\sigma \in \Sigma$ one of its states.*

1. *If $\sigma \rightarrow_{tick} \sigma'$, then $[\![t]\!]_\sigma \neq on(0)$, for all timers t.*
2. *If $\sigma \rightarrow_{tick} \sigma'$, then for all channel states (c, q), $q = \epsilon$.*

Proof. For part (1), if $[\![t]\!]_\eta = on(0)$ for a timer t in a process P, then a τ-step is allowed due to either Timeout or TDiscard of Table 1. Hence, the system is not *blocked* and therefore cannot do a *tick*-step.

Part (2) follows from the fact that a channel can only perform a *tick*-step exactly when it is empty. □

The following lemma expresses that the blocked predicate is compositional in the sense that the parallel composition of processes is blocked iff each process is blocked (cf. rule Tick of Table 3).

Lemma 2. *For a state $\sigma = (\sigma_1, \ldots, \sigma_n)$ of a system S, blocked(σ) iff blocked(σ_i) for all σ_i.*

Proof. If σ is not blocked, it can perform a τ-step or an output-step. The output step must originate from a process, which thus is not blocked. The τ-step is either caused by a single process or by a synchronizing action of a sender and a receiver; in both cases at least one process is not blocked. For the reverse direction, a τ-step of a single process being thus not blocked, entails that σ is not blocked. An output step of a single process causes σ either to do the same output step or, in case of internal communication, to do a τ-step. In both cases, σ is not blocked. □

3 Replacing asynchronous with synchronous communication

In a system with asynchronous communication, introducing an environment process can lead to a combinatorial explosion caused by all combinations of messages in the queues modelling channels. An ideal solution from the perspective of the state space would be to construct an environment process that communicates with the system synchronously. In this section, we specify under which conditions we can safely replace the asynchronous communication with an outside environment process, say E, by *synchronous* communication.

A general condition an asynchronously communicating process satisfies is that the process is always willing to accept messages, since the queues are unbounded. Hence, the environment process must be at least *input enabled:* it must always be able to receive messages, lest the synchronous composition will lead to more blockings. Thanks to the DISCARD-rule of Table 1, SDL-processes are input enabled, i.e., at least *input-discard* steps are possible, which throw away the message and do not change the state of the process. Another effect of an input queue is that the queue introduces an arbitrary delay between the reception of a message and the future reaction of the receiving process to this message. For an output, the effect is converse. This means, that in order not to miss any potential behavior, the asynchronous process can be replaced by the analogous synchronous process as long as there are either only input actions or else only output actions, so the process is not reactive.[4] This is related to the so-called *Brock-Ackerman anomaly*, characterizing the difference between buffered and unbuffered communication [7].

Disallowing reactive behavior is clearly a severe restriction and only moderately generalizes completely chaotic behavior. One feature of the timed semantics, though, allows to loosen this restriction. Time progresses by *tick*-steps when the system is blocked. This especially means that when a *tick* happens, all queues of a system are empty (cf. Lemma 1). This implies that the restrictions need to apply only *per time slice,* i.e., to the steps between two ticks,[5] and not to the overall process behavior. Additionally we require that there are no infinite sequences of steps without a *tick*, i.e., there are no runs with *zero-time cycles.* This leads to the following definition.

Definition 3. *A sequence of steps is* tick-separated *iff it contains no zero-time cycle, and for every time slice of the sequence one of the following two conditions holds:*

1. *the time slice contains no output action;*

[4] A more general definition would require that the process actions satisfy a *confluence* condition as far as the input and output actions are concerned, i.e., doing an input action does not invalidate the possibility of an output action, and vice versa. Also in this case, the process is not reactive, since there is no feed-back from input to output actions.

[5] A time slice of a run is a maximal subsequence of the run without *tick*-steps.

$$\frac{\gamma_1 \rightarrow_{c_i?(s,v)} \hat{\gamma}_1 \qquad \gamma_2 \rightarrow_{c_o!(s,v)} \hat{\gamma}_2}{(\gamma_1, \gamma_2) \rightarrow_\tau (\hat{\gamma}_1, \hat{\gamma}_2)} \ \textsc{Comm}_{sync}$$

Table 4. Synchronous communication over rendezvous channel c

2. the time slice contains no output over two different channels, and all locations in the time slice are input-discarding wrt. all inputs of that time slice.

We call a process tick-separated *if all its runs are* tick*-separated.*

Further we consider a synchronous version P_s and an asynchronous version P_a of a process P, where P_s is the process P together with a set of rendezvous channels, and P_a is formed by the process P together with a set of channels with the same names as for P_s but which are queues. Synchronous communication over a rendezvous channel c is defined by rule $\textsc{Comm}_{sync}$ of Table 4.

In the following, we call a configuration the combined state of a process together with the state of its channels. So given P_s and P_a and two corresponding states $\gamma_s = \sigma_s$ and $\gamma_a = (\sigma_a, (c_i, q_i), (c_o^1, q_1), \ldots, (c_o^k, q_k))$, we define $\trianglerighteq$ as $\gamma_a \trianglerighteq \gamma_s$, if $\sigma_s = \sigma_a$. Comparing the observable behavior of an asynchronous and a synchronous process, we must take into account that the asynchronous one performs more internal steps when exchanging messages with its queues. Hence the comparison is based on a *weak* notion of equivalence, ignoring the τ-steps: so we define a weak step $\Rightarrow_\lambda$ as $\rightarrow_\tau^*\rightarrow_\lambda\rightarrow_\tau^*$ when $\lambda \neq \tau$, and as $\rightarrow_\tau^*$ else. Correspondingly, $\Rightarrow_{\vec{\lambda}}$ denotes a sequence of weak steps with labels from a sequence $\vec{\lambda}$.

Lemma 4. *Assume a synchronous and an asynchronous version P_s and P_a of a process P and corresponding states γ_s and γ_a with $\gamma_a \trianglerighteq \gamma_s$, where the queues of γ_a are all empty. If $\gamma_a \Rightarrow_{\vec{\lambda}} \gamma_a'$ by a* tick*-separated sequence, where $\vec{\lambda}$ does not contain a tick-label, and where the queues of γ_a' are empty, then there exists a sequence $\gamma_s \Rightarrow_{\vec{\lambda}} \gamma_s'$ with $\gamma_a' \trianglerighteq \gamma_s'$.*

Proof. We are given a sequence $\gamma_a = \gamma_0^a \rightarrow_{\lambda_0} \gamma_1^a \ldots \rightarrow_{\lambda_{n-1}} \gamma_n^a = \gamma_a'$, with the queues of γ_0^a and γ_n^a empty. According to the definition of *tick*-separation, we distinguish the following two cases:

Case 1: $\lambda_i \notin \{tick, c!(s,v)\}$, for all $1 \leq i \leq n-1$

To get a matching reduction sequence of the synchronous system starting at γ_0^s, we apply the following renaming scheme. Input actions $\gamma_a \rightarrow_{c?(s,v)} \gamma_a'$ into the queue are just omitted (which means, they are postponed for the synchronous process). τ-steps $\gamma_a \rightarrow_\tau \gamma_a'$, inputting a value from the queue into the process, i.e., τ-steps justified by rule $\textsc{Comm}$ where the process does a step $\sigma \rightarrow_{c?(s,v)} \sigma'$ by rule $\textsc{Input}$ and the queue the corresponding output step by rule $\textsc{Out}$, are replaced by a direct input steps $\gamma_s \rightarrow_{c?(s,v)} \gamma_s'$. Process internal τ-steps of the asynchronous system are identically taken by the synchronous system, as well.

τ-steps caused by output actions from the process into a queue need not be dealt with, since the sequence from γ_0^a to γ_n^a does not contain external output from the queues, and the queues are empty at the beginning and the end of the sequence.

It is straightforward to see that the sequence of steps obtained by this transformation is indeed a legal sequence of the synchronous system. Moreover, the last configurations have the same state component and, due to the non-lossiness and the Fifo-behavior of the input queue, both sequences coincide modulo τ-steps.

Case 2: no output over two different channels, input discarding locations (and no *tick*-steps)
Similar to the previous case, the synchronous system can mimic the behavior of the asynchronous one adhering to the following scheme: τ-steps $\gamma_a \rightarrow_\tau \gamma'_a$, feeding a value from the process into the queue, i.e., τ-steps justified by rule OUTPUT where the process does a step $\sigma \rightarrow_{c!(s,v)} \sigma'$ and the queue the corresponding input step by rule IN, are replaced by a direct output step $\gamma_s \rightarrow_{c!(s,v)} \gamma'_s$. Input actions $\gamma_a \rightarrow_{c?(s,v)} \gamma'_a$ into the queue are mimicked by a discard-step. Output steps from the queue of the asynchronous system are omitted, and so are τ-steps caused by internal communication from the input-queue to the process. All other internal steps are identically taken in both systems. The rest of the argument is analogous to the previous case. □

Note that $\gamma'_a \trianglerighteq \gamma'_s$ means that γ'_s is blocked whenever γ'_a is blocked.

We write $[\![P]\!]_{wtrace}$ to denote the set of all weak traces of process P. To prove that for *tick*-separated processes $[\![P_s]\!]_{wtrace} = [\![P_a]\!]_{wtrace}$, we introduce a notion of *tick*-simulation that captures the ability to simulate any *sequence* of steps up to a *tick* step, i.e. the chosen granularity level are time slices and only the states immediately before and after *tick* are of importance there. (Remember that we assume the absence of zero-time cycles.)

Definition 5. *A binary relation $\mathcal{R} \subseteq \Gamma_1 \times \Gamma_2$ on two sets of states is called a* tick-*simulation, when the following conditions hold:*

1. *If $\gamma_1 \mathcal{R} \gamma_2$ and $\gamma_1 \rightarrow_{tick} \gamma'_1$, then $\gamma_2 \rightarrow_{tick} \gamma'_2$ and $\gamma'_1 \mathcal{R} \gamma'_2$.*
2. *If $\gamma_1 \mathcal{R} \gamma_2$ and $\gamma_1 \Rightarrow_{\vec{\lambda}} \gamma'_1$ for some γ'_1 with blocked(γ'_1) where $\vec{\lambda}$ does not contain tick, then $\gamma_2 \Rightarrow_{\vec{\lambda}} \gamma'_2$ for some γ'_2 with blocked(γ'_2) and $\gamma'_1 \mathcal{R} \gamma'_2$.*

We write $\gamma_1 \preceq_{tick} \gamma_2$ if there exists a tick *simulation $\mathcal{R}$ with $\gamma_1 \mathcal{R} \gamma_2$, and similarly for processes, $P_1 \preceq_{tick} P_2$ if their initial states are in that relation.*

Theorem 6. *If a process P is* tick-*separated, then $[\![P_s]\!]_{wtrace} = [\![P_a]\!]_{wtrace}$.*

Proof. There are two directions to show. $[\![P_s]\!]_{wtrace} \subseteq [\![P_a]\!]_{wtrace}$ is immediate: each communication step of the synchronous process P_s can be mimicked by the buffered P_a with adding an internal τ-step for the communication with the buffer.

For the reverse direction $[\![P_a]\!]_{wtrace} \subseteq [\![P_s]\!]_{wtrace}$ we show that P_a is simulated by P_s according to the definition of *tick*-simulation, which considers as basic steps only *tick*-steps or else the sequence of steps within one time slice.

We define the relation $\mathcal{R} \subseteq \Gamma_a \times \Gamma_s$ as $(\sigma_a, ((c_0, q_0), \ldots, (c_m, q_m)))\mathcal{R}\sigma_s$ iff $\sigma_a = \sigma_s$ and $q_i = \epsilon$ for all queues modelling the channels. To show that $\mathcal{R}$ is indeed a *tick*-simulation, assume $\gamma_a = (\sigma_a, ((c_0, \epsilon), \ldots, (c_m, \epsilon)))$ and $\gamma_s = \sigma_s$ with $\gamma_a \mathcal{R} \gamma_s$. There are two cases to consider.

Case: $\gamma_a \rightarrow_{tick} \gamma'_a$
where $\gamma'_a = \gamma_{a[t \mapsto (t-1)]}$. By the definition of the *tick*-step, *blocked*(γ_a) must hold, i.e., there are no steps enabled except input from the outside or *tick*-steps. Since immediately *blocked*(γ_s), also $\gamma_s \rightarrow_{tick} \gamma_{s[t \mapsto (t-1)]}$, which concludes the case.

Case: $\gamma_a \Rightarrow_{\vec{\lambda}} \gamma'_a$
where *blocked*(γ'_a) and $\vec{\lambda}$ does not contain a *tick*-label. The case follows directly from Lemma 4 and the fact that $\gamma'_a \trianglerighteq \gamma'_s$ where γ'_a is blocked implies that also γ'_s is blocked.

Since clearly the initial states are in relation $\mathcal{R}$ as defined above, this gives $P_a \preceq_{tick} P_s$. Since $P_a \preceq_{tick} P_s$ and each *tick*-step of P_a can be mimicked by the *tick* step of P_s and each weak step $\Rightarrow_{\vec{\lambda}}$ of P_a can also be mimicked by P_s. That implies $[\![P_a]\!]_{wtrace} \subseteq [\![P_s]\!]_{wtrace}$, as required. □

4 Abstracting data

Originating from an unknown or underspecified environment, signals from outside can carry *any* value, which renders the system infinite state. Assuming nothing about the data means one can conceptually abstract values from outside into one abstract *"chaotic"* value, which basically means to ignore these data and focus on the control structure. Data not coming from outside is left untouched, though chaotic data from the environment influence internal data of the system. In this section, we present a straightforward dataflow analysis marking variable and timer instances that may be influenced by the environment, namely we establish for each process- and timer-variable in each location whether

1. the variable is guaranteed to be non-chaotic, or
2. the variable is guaranteed to be influenced by the outside, or
3. whether its status depends on the actual run.

The analysis is a combination of the ones from [29] and [17].

4.1 Dataflow analysis

The analysis works on a simple *flow graph* representation of the system, where each process is represented by a single flow graph, whose nodes $n \in$ *nodes* are associated with the process' actions and the flow relation captures the intra-process data dependencies. Since the structure of the language we consider is rather simple, the flow-graph can be easily obtained by standard techniques.

We use an abstract representation of the data values, where $\top$ is interpreted as value chaotically influenced by the environment and $\perp$ stands for a non-chaotic value. We write $\eta^\alpha, \eta_1^\alpha, \ldots$ for abstract valuations, i.e., for typical elements from $Val^\alpha = Var \to \{\top, \bot\}$. The abstract values are ordered $\bot \leq \top$, and the order is lifted pointwise to valuations. With this ordering, the set of valuations forms a complete lattice, where we write $\eta_\bot$ for the least element, given as $\eta_\bot(x) = \bot$ for all $x \in Var$, and we denote the least upper bound of $\eta_1^\alpha, \ldots, \eta_n^\alpha$ by $\bigvee_{i=1}^n \eta_i^\alpha$ (or by $\eta_1^\alpha \vee \eta_2^\alpha$ in the binary case). By slight abuse of notation, we will use the same symbol η^α for the valuation per node, i.e., for functions of type $node \to Val^\alpha$. The abstract valuation $[\![e]\!]_{\eta^\alpha}$ for an expression e equals $\bot$ iff all variables in e are evaluated to $\bot$, $[\![e]\!]_{\eta^\alpha}$ is $\top$ iff the abstract valuation of at least one of the variables in e is $\top$.

Depending on whether we are interested in an answer to point (1) or point (2) from above, $\top$ is interpreted as a variable potentially influenced from outside, and, dually for the second case, $\top$ stands for variables guaranteed to be influenced from outside. Here we present *may* and *must* analysis for the first and the second case respectively.

May analysis First we consider *may* analysis that marks variables *potentially* influenced by data from outside. Each node n of the flow graph has associated an abstract transfer function $f_n : Val^\alpha \to Val^\alpha$, describing the change of the abstract valuations depending on the kind of action at the node. The functions are given in Table 5. The equations are mostly straightforward; the only case deserving mention is the one for $c?s(x)$, whose equation captures the inter-process data-flow from a sending to a receiving action. It is easy to see that the transfer functions are *monotone.*

$$
\begin{aligned}
f(c?s(x))\eta^\alpha &= \begin{cases} \eta^\alpha[x \mapsto \top] & s \in Sig_{ext} \\ \eta^\alpha[x \mapsto \bigvee\{[\![e]\!]_{\eta^\alpha} \mid n' = g \rhd c!s(e)\}] & s \notin Sig_{ext} \end{cases} \\
f(g \rhd c!s(e))\eta^\alpha &= \eta^\alpha \\
f(g \rhd x := e)\eta^\alpha &= \eta^\alpha[x \mapsto [\![e]\!]_{\eta^\alpha}] \\
f(g \rhd set\ t := e)\eta^\alpha &= \eta^\alpha[t \mapsto on([\![e]\!]_{\eta^\alpha})] \\
f(g \rhd reset\ t)\eta^\alpha &= \eta^\alpha[t \mapsto off] \\
f(g_t \rhd reset\ t)\eta^\alpha &= \eta^\alpha[t \mapsto off]
\end{aligned}
$$

Table 5. May analysis: transfer functions/abstract effect for process P

Upon start of the analysis, at each node the variables' values are assumed to be defined, i.e., the initial valuation is the least one: $\eta_{init}^\alpha(n) = \eta_\bot$. This choice rests on the assumption that all local variables of each process are properly initialized. We are interested in the least solution to the data-flow problem given by the following constraint set:

$$\begin{aligned} \eta^{\alpha}_{post}(n) &\geq f_n(\eta^{\alpha}_{pre}(n)) \\ \eta^{\alpha}_{pre}(n) &\geq \bigvee \{\eta^{\alpha}_{post}(n') \mid (n', n) \text{ in flow relation}\} \end{aligned} \tag{1}$$

For each node n of the flow graph, the data-flow problem is specified by two inequations or constraints. The first one relates the abstract valuation η^{α}_{pre} before entering the node with the valuation η^{α}_{post} afterwards via the abstract effects of Table 5. The *least* fixpoint of the constraint set can be found iteratively in a fairly standard way by a *worklist algorithm* (see e.g., [19,14,24]), where the worklist steers the iterative loop until the least fixpoint is reached (cf. Figure 1).

```
input  : the flow−graph of the program
output : η^α_pre, η^α_post ;

η^α(n) = η^α_init(n) ;
WL = {n | α_n = ?s(x), s ∈ Sig_ext} ;

repeat
      pick n ∈ WL ;
      if    n = g ▷ c!s(e)  then
            let    S′ = {n′ | n′ = c?s(x) and ⟦e⟧_η^α(n) ⋢ ⟦x⟧_η^α(n′)}
            in
                  for all n′ ∈ S′ :  η^α(n′) := f_n′(η^α(n′)) ;
      let  S = {n″ ∈ succ(n) | f_n(η^α(n)) ⋢ η^α(n″)}
      in
            for all n″ ∈ S :  η^α(n″) := f_n(η^α(n)) ;
      WL := WL\{n} ∪ S ∪ S′ ;
until  WL = ∅ ;

η^α_pre(n) = η^α(n) ;
η^α_post(n) = f_n(η^α(n))
```

Fig. 1. *May* analysis: worklist algorithm

The worklist data-structure WL used in the algorithm is a set of elements, more specifically a set of nodes from the flow-graph, where we denote by $succ(n)$ the set of successor nodes of n in the flow graph in forward direction. It supports as operation to randomly pick one element from the set (without removing it), and we write $WL\backslash\{n\}$ for the worklist without the node n and $\cup$ for set-union on the elements of the worklist. The algorithm starts with the least valuation on all nodes and an initial worklist containing nodes with input from the environment. It enlarges the valuation within the given lattice step by step until it stabilizes, i.e., until the worklist is empty. If adding the abstract effect of one node to the current state enlarges the valuation, i.e., the set S is non-empty, those successor nodes from S are (re-)entered into the list of unfinished one. Since the set of variables in the system is finite, and thus the lattice of abstract valuations, the termination of the algorithm is immediate.

With the worklist as a set-like data structure, the algorithm is free to work off the list in any order. In praxis, more deterministic data-structures and traversal strategies are appropriate, for instance traversing the graph in a breadth-first manner (see [24] for a broader discussion or various traversal strategies).

After termination the algorithm yields two mappings $\eta^\alpha_{pre}, \eta^\alpha_{post} : Node \to Val^\alpha$. On a location l, the result of the analysis is given by $\eta^\alpha(l) = \bigvee\{\eta^\alpha_{post}(\tilde{n}) \mid \tilde{n} = \tilde{l} \longrightarrow_\alpha l\}$, also written as η^α_l.

Lemma 7 (Correctness (*may*)). *Upon termination, the algorithm gives back the least solution to the constraint set as given by the equations (1), resp. Table 5.*

Must analysis The *must* analysis is almost dual to *may* analysis. A transfer function that describes the change of the abstract valuation depending on the action at the node is defined in Table 6. For inputs, $c?s(x)$ in process P assigns $\perp$ to x if the signal is sent to P with reliable data, only. This means the values after reception correspond to the greatest lower bound over all expressions which can occur in a matching send-action.

$$
\begin{aligned}
f(c?s(x))\eta^\alpha &= \begin{cases} \eta^\alpha{}_{[x \mapsto \top]} & s \notin Sig_{int} \\ \eta^\alpha{}_{[x \mapsto \bigwedge\{[\![e]\!]_{\eta^\alpha} \mid n' = g \,\triangleright\, c!s(e)\}]} & s \in Sig_{int} \end{cases} \\
f(g \triangleright c!s(e))\eta^\alpha &= \eta^\alpha \\
f(g \triangleright x := e)\eta^\alpha &= \eta^\alpha{}_{[x \mapsto [\![e]\!]_{\eta^\alpha}]} \\
f(g \triangleright set\ t := e)\eta^\alpha &= \eta^\alpha{}_{[t \mapsto on([\![e]\!]_{\eta^\alpha})]} \\
f(g \triangleright reset\ t)\eta^\alpha &= \eta^\alpha{}_{[t \mapsto off]} \\
f(g_t \triangleright reset\ t)\eta^\alpha &= \eta^\alpha{}_{[t \mapsto off]}
\end{aligned}
$$

Table 6. Must analysis: transfer functions/abstract effect for process P

As that is done for may analysis, the data-flow problem is specified for each node n of the flow graph by two inequations (2) (see Table 6). Analogously, the *greatest* fixpoint of the constraint set can be found iteratively by a worklist algorithm (cf. Figure 2). Upon start of the analysis, at each node the variables' values are assumed to be defined, i.e., the initial valuation is the greatest one: $\eta^\alpha_{init}(n) = \eta_\top$.

$$
\begin{aligned}
\eta^\alpha_{post}(n) &\leq f_n(\eta^\alpha_{pre}(n)) \\
\eta^\alpha_{pre}(n) &\leq \bigwedge\{\eta^\alpha_{post}(n') \mid (n', n) \text{ in flow relation}\}
\end{aligned}
\tag{2}
$$

```
input:   the flow-graph of the program
output:  η^α_pre, η^α_post;

η^α(n) = η^α_init(n);
WL = {n | α_n = g ▷ x := e};

repeat
    pick n ∈ WL;
    if n = g ▷ c!s(e) then
        let  S' = {n' | n' = c?s(x) and [[e]]_{η^α(n)} ⋧ [[x]]_{η^α(n')}}
        in
            for all n' ∈ S': η^α(n') := f_{n'}(η^α(n'));
    let S = {n'' ∈ succ(n) | f_n(η^α(n)) ⋧ η^α(n'')}
    in
        for all n'' ∈ S: η^α(n'') := f_n(η^α(n));
    WL := WL\{n} ∪ S ∪ S';
until WL = ∅;

η^α_pre(n) = η^α(n);
η^α_post(n) = f_n(η^α(n))
```

Fig. 2. *Must* analysis: worklist algorithm

Like the may-analysis case, the termination of the algorithm follows from the finiteness of the set of variables.

Lemma 8 (Correctness (*must*)). *Upon termination, the algorithm from Figure 2 gives back the greatest solution to the constraint set as given by equations (2) resp. Table 6.*

4.2 Program transformation

Based on the result of the analysis, we transform the given system $S = P \parallel \bar{P}$ into an optimized one, denoted by $S^\sharp$, where the communication of P with its environment $\bar{P}$ is done synchronously, all the data exchanged is abstracted, and which is in a simulation relation with the original system.

The intention is to use the information collected in the analyses about the influence of the environment to reduce the state space. Depending on whether one relies on the may-analysis alone (which variable occurrences may be influenced from the outside) or takes into account the results of both analyses (additional information which variable occurrences are definitely chaotic) the precision of the abstraction varies. Using only the may-information overapproximates the system (further) but in general leads to a smaller state space.

The second option, on the other hand, gives a more precise abstraction and thus less false negatives. Indeed, it does not, apart from the abstraction caused by introducing chaotic values, abstract the system further as far as the behavior is concerned. It is nevertheless profitable as it allows to remove any unnecessary instances of variables or expressions which are detected to be $\mathbb{T}$ constantly. It furthermore can make further optimizations of the system more effective. For instance, live analysis and the optimization as described in [4] can be effective for more variable instances and thus yield better further reduction when applied after replacing variable instances which are constantly $\mathbb{T}$.

In either case we must ensure that the abstraction of timer values is treated adequately (see below). Here we describe the transformation for the combination of may and must analysis, only, since the alternative is simpler.

Overloading the symbols $\top$ and $\bot$ we mean for the rest of the paper: the value of $\top$ for a variable at a location refers to the result of the must analysis, i.e., the definite knowledge that the data is chaotic for all runs. Dually, $\bot$ stands for the definite knowledge of the may analysis, i.e., for data which is never influenced from outside. Additionally, we write $\mathbb{I}$ in case neither analysis gave a definite answer.

We extend the data domains each by an additional value $\mathbb{T}$, representing unknown, chaotic, data, i.e., we assume now domains such as $\mathbb{N}^{\mathbb{T}} = \mathbb{N} \,\dot{\cup}\, \{\mathbb{T}\}$, $\mathit{Bool}^{\mathbb{T}} = \mathit{Bool} \,\dot{\cup}\, \{\mathbb{T}\}, \ldots$, where we do not distinguish notationally the various types of chaotic values. These values $\mathbb{T}$ are considered as the largest values, i.e., we introduce $\leq$ as the smallest reflexive relation with $v \leq \mathbb{T}$ for all elements v (separately for each domain). The strict lifting of a valuation $\eta^{\mathbb{T}}$ to expressions is denoted by $[\![.]\!]_{\eta^{\mathbb{T}}}$.

The transformation is straightforward: guards influenced by the environment are taken non-deterministically, i.e., a guard g at a location l is replaced by *true*, if $[\![g]\!]_{\eta_l^\alpha} = \top$. A guard g whose value at a location l is $\mathbb{I}$ is treated dynamically on the extended data domain. For assignments, we distinguish between the variables that carry the value $\mathbb{I}$ in at least one location and the rest. Assignments of $\mathbb{T}$ to variables that take $\mathbb{I}$ at no location are omitted. Assignments of concrete values are left untouched and the assignments to the variables that are marked by $\mathbb{I}$ in at least one location are performed on the extended data domain.

The interpretation of *timer variables* on the extended domain requires special attention. Chaos can influence timers only via the *set*-operation by setting it to a chaotic value. Therefore, the domain of timer values contains the additional chaotic value $\mathit{on}(\mathbb{T})$. Since we need the transformed system to show at least the behavior of the original one, we must provide proper treatment of the rules involving $\mathit{on}(\mathbb{T})$, i.e., the TIMEOUT-, the TDISCARD-, and the TICK-rule. As $\mathit{on}(\mathbb{T})$ stands for any value of active timers, it must cover the cases where timeouts and timer-discards are enabled (because of the concrete value $\mathit{on}(0)$) as well as *disabled* (because of $\mathit{on}(n)$ with $n \geq 1$). The second one is necessary, since the enabledness of the tick steps depends on the disabledness of timeouts and timer discards via the blocked-condition.

To distinguish the two cases, we introduce a refined abstract value $\mathit{on}(\mathbb{T}^+)$ for chaotic timers, representing all *on*-settings larger than or equal to 1. The order on the domain of timer values is given as smallest reflexive order relation

$$\frac{l \longrightarrow_{c?s(x)} \hat{l} \in Edg \qquad x \notin Var_{\mathbb{T}} \qquad [\![x]\!]_{\eta_l^\alpha} = \top}{l \longrightarrow_{c?s(\_)} \hat{l} \in Edg^\sharp} \text{ T-INPUT}_{ext}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, c!(s,e)} \hat{l} \in Edg \qquad [\![e]\!]_{\eta_l^\alpha} = \top}{l \longrightarrow_{g^\sharp \,\triangleright\, c!(s,\mathbb{T})} \hat{l} \in Edg^\sharp} \text{ T-OUTPUT}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, x:=e} \hat{l} \in Edg \qquad x \notin Var_{\mathbb{T}} \qquad [\![x]\!]_{\eta_l^\alpha} = \top}{l \longrightarrow_{g^\sharp \,\triangleright\, skip} \hat{l} \in Edg^\sharp} \text{ T-ASSIGN}_1$$

$$\frac{l \longrightarrow_{g \,\triangleright\, x:=e} \hat{l} \in Edg \qquad x \in Var_{\mathbb{T}} \qquad [\![e]\!]_{\eta_l^\alpha} = \top}{l \longrightarrow_{g^\sharp \,\triangleright\, x:=\mathbb{T}} \hat{l} \in Edg^\sharp} \text{ T-ASSIGN}_2$$

$$\frac{l \longrightarrow_{g \,\triangleright\, set\ t:=e} \hat{l} \in Edg \qquad [\![e]\!]_{\eta_l^\alpha} = \top}{l \longrightarrow_{g^\sharp \,\triangleright\, set\ t:=\mathbb{T}} \hat{l} \in Edg^\sharp} \text{ T-SET}$$

Table 7. Transformation

such that $on(0) \leq on(\mathbb{T})$ and $on(n) \leq on(\mathbb{T}^+) \leq on(\mathbb{T})$, for all $n \geq 1$. To treat the case where the abstract timer value $on(\mathbb{T})$ denotes *absence* of immediate timeout, we add edges of the form

$$\frac{}{l \longrightarrow_{t=on(\mathbb{T}) \,\triangleright\, set\ t:=\mathbb{T}^+} l \in Edg^\sharp} \text{ T-NOTIMEOUT}$$

which set back the timer value to $\mathbb{T}^+$ representing a non-zero delay.

The decreasing operation needed in the TICK-rule is defined in extension to the definition on values from $on(\mathbb{N})$ on $\mathbb{T}^+$ by $on(\mathbb{T}^+) - 1 = on(\mathbb{T})$. Note that the operation is left undefined on $\mathbb{T}$, which is justified by a property analogous to Lemma 1:

Lemma 9. *Let $(l, \eta^{\mathbb{T}}, q^{\mathbb{T}})$ be a state of $S^\sharp$. If $(l, \eta^{\mathbb{T}}, q^{\mathbb{T}}) \rightarrow_{tick}$, then $[\![t]\!]_{\eta^{\mathbb{T}}} \notin \{on(\mathbb{T}), on(0)\}$, for all timers t.*

Proof. By definition of the *blocked*-predicate and inspection of the TIMEOUT- and TDISCARD-rule (for $on(0)$ as for Lemma 1) and the behavior of the abstract value $on(\mathbb{T})$ (T-NOTIMEOUT-rule). □

As the transformation only adds non-determinism, the transformed system $S^\sharp$ simulates S (cf. [29]). Together with Theorem 6, this guarantees preservation of LTL-properties as long as variables influenced by $\bar{P}$ are not mentioned. Since we abstracted external data into a single value, not being able to specify properties depending on externally influenced data is not much of an additional loss of precision.

Theorem 10. *Let S_a and S_s be the variant of a system communicating to the environment asynchronously resp. synchronously, and S be given as the parallel* *composition* $S_a \parallel \bar{S}$*, where* $\bar{S}$ *is the environment of the system. Furthermore, let* $S^\sharp = S_s^\sharp \parallel \bar{S}$ *be defined as before, and* φ *a next-free LTL-formula mentioning only variables from* $\{x \mid \forall l \in Loc.\ [\![x]\!]_{\eta_l^\alpha} = \bot\}$*. Then* $S^\sharp \models \varphi$ *implies* $S \models \varphi$*.*

5 Example: a wireless ATM medium-access protocol

The goal of our experiments is to estimate the state space reduction due to replacing asynchronous communication with the environment by the synchronous one. Primarily interested in the effect of removing queues, we use here the most trivial environment: the *chaotic* one.

We applied the methods in a series of experiments to the industrial protocol Mascara [33]. Located between the ATM-layer and the physical medium, Mascara is a medium-access layer or, in the context of the ISDN reference model, a transmission convergence sub-layer for wireless ATM communication in local area networks. A crucial feature of Mascara is the support of *mobility.* A mobile terminal (MT) located inside the area cell of an access point (AP) is capable of communicating with it. When a mobile terminal moves outside the current cell, it has to perform a so-called *handover* to another access point covering the cell the terminal has moved into. The handover must be managed transparently with respect to the ATM layer, maintaining the agreed quality of service for the current connections. So the protocol has to detect the need for a handover, select a candidate AP to switch to, and redirect the traffic with minimal interruption.

This protocol was the main case study in the Vires project; the results of its verification can be found e.g. in [3,13,30]. The SDL-specification of the protocol was automatically translated into the input language of DTSpin [2,11], a discrete-time extension of the well-known Spin model-checker [16]. For the translation, we used sdl2if [5] and if2pml-translators [3]. Our prototype implementation, the pml2pml-translator, post-processes the output from the automatic translation of the SDL-specification into DTPromela.

Here, we are not interested in Mascara itself and the verification of its properties, but as real-life example for the comparison of the state spaces of parts of the protocol when closed with the environment as an asynchronous chaotic process and the state space of the same entity closed with embedded chaos. For the comparison we chose a model of the *Mascara control* entity (MCL) at the mobile terminal side In our experiments we used DTSpin version 0.1.1, an extension of Spin3.3.10, with the partial-order reduction and compression options on. All the experiments were run on a Silicon Graphics Origin 2000 server on a single R10000/250MHz CPU with 8GB of main memory.

The implementation currently covers the may analysis and the corresponding transformation. We do not model the chaotic environment as a separate process communicating with the system via rendezvous channels but transform an open DTPromela model into a closed one by embedding the timed chaotic environment into the system as described in [29], which allows us to use the process fairness mechanism provided by Spin, which works only for systems with asynchronous communication. The translator does not require any user interaction, except that the user is requested to give the list of external signals. The extension is implemented in Java and requires JDK-1.2 or later. The package can be downloaded from `http://www.cwi.nl/~ustin/EH.html`.

bs	states	transitions	mem.	time	states	transitions	mem.	time
2	9.73e+05	3.64e+06	40.842	15:57	300062	1.06e+06	9.071	1:13
3	5.24e+06	2.02e+07	398.933	22:28	396333	1.85e+06	11.939	1:37
4	2.69e+07	1.05e+08	944.440	1:59:40	467555	2.30e+06	14.499	2:13

Table 8. Model checking MCL with chaos as a process and embedded chaos

Table 5 gives the results for the model checking of MCL with chaos as external process on the left and embedded on the right. The first column gives the buffer size for process queues. The other columns give the number of states, transitions, memory and time consumption, respectively. As one can see, the state space as well as the time and the memory consumption are significantly larger for the model with the environment as a process, and they grow with the buffer size much faster than for the model with embedded chaos. The model with the embedded environment has a relatively stable state-space size.

6 Conclusion

In this paper, we integrated earlier works from [29,18,31,17] into a general framework describing how to close an open, asynchronous system by a timed environment while avoiding the combinatorial state-explosion in the external buffers. The generalization presented here goes a step beyond complete arbitrary environmental behavior, using the timed semantics of the language. We facilitate the model checking of the system by using the information obtained with may and must analyses: We substitute the chaotic value $\mathbb{T}$ for expressions influenced by chaotic data from outside and then optimize the system by removing variables and actions that became redundant.

In the context of software-testing, [9] describes an a dataflow algorithm to close program fragments given in the C-language with the most general environment. The algorithm is incorporated into the *VeriSoft* tool. As in our paper, the assume an asynchronous communicating model and abstract away external data, but do not consider *timed* systems and their abstraction. As for model-checking and analyzing SDL-programs, much work has been done, for instance in the context of the Vires-project, leading to the IF-toolset [5]

A fundamental approach to model checking open systems is known as *module* checking [21][20]. Instead of transforming the system into a closed one, the underlying computational model is generalized to distinguish between transitions under control of the module and those driven by the environment. Mocha [1] is a model checker for reactive modules, which uses alternating-time temporal logic as specification language.

For practical applications, we are currently extending the larger case study [30] using the chaotic closure to this more general setting. We proceed in the following way: after splitting an SDL system into subsystems following the system structure, properties of the subsystems are verified being closed with an embedded chaotic environment. Afterwards, the verified properties are encoded into an SDL process, for which a tick-separated closure is constructed. This closure is used as environment for other parts of the system. As the closure gives a safe abstraction of the desired environment behavior, the verification results can be transferred to the original system.

References

1. R. Alur, T. A. Henzinger, F. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. Mocha: Modularity in model checking. In A. J. Hu and M. Y. Vardi, editors, *Proceedings of CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, pages 521–525. Springer-Verlag, 1998.
2. D. Bošnački and D. Dams. Integrating real time into Spin: A prototype implementation. In S. Budkowski, A. Cavalli, and E. Najm, editors, *Proceedings of Formal Description Techniques and Protocol Specification, Testing, and Verification (FORTE/PSTV'98)*. Kluwer Academic Publishers, 1998.
3. D. Bošnački, D. Dams, L. Holenderski, and N. Sidorova. Verifying SDL in Spin. In S. Graf and M. Schwartzbach, editors, *TACAS 2000*, volume 1785 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
4. M. Bozga, J. C. Fernandez, and L. Ghirvu. State space reduction based on Live. In A. Cortesi and G. Filé, editors, *Proceedings of SAS '99*, volume 1694 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
5. M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm, and L. Mounier. IF: An intermediate representation and validation environment for timed asynchronous systems. In J. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of Symposium on Formal Methods (FM 99)*, volume 1708 of *Lecture Notes in Computer Science*. Springer-Verlag, Sept. 1999.
6. M. Bozga, S. Graf, A. Kerbrat, L. Mounier, I. Ober, and D. Vincent. SDL for real-time: What is missing? In Y. Lahav, S. Graf, and C. Jard, editors, *Electronic Proceedings of SAM'00*, 2000.
7. J. Brock and W. Ackerman. An anomaly in the specifications of nondeterministic packet systems. Technical Report Computation Structures Group Note CSG-33, MIT Lab. for Computer Science, Nov. 1977.
8. E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994. A preliminary version appeared in the Proceedings of POPL 92.
9. C. Colby, P. Godefroid, and L. J. Jagadeesan. Automatically closing of open reactive systems. In *Proceedings of 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 1998.
10. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems: Abstraction preserving $\forall$CTL*,$\exists$CTL*, and CTL*. In E.-R. Olderog, editor, *Proceedings of PROCOMET '94*. IFIP, North-Holland, June 1994.

11. Discrete-time Spin. `http://www.win.tue.nl/~dragan/DTSpin.html`, 2000.
12. P. Godefroid. Using partial orders to improve automatic verification methods. In E. M. Clarke and R. P. Kurshan, editors, *Computer Aided Verification 1990*, volume 531 of *Lecture Notes in Computer Science*, pages 176–449. Springer-Verlag, 1991. an extended Version appeared in ACM/AMS DIMACS Series, volume 3, pages 321–340, 1991.
13. J. Guoping and S. Graf. Verification experiments on the Mascara protocol. In M. B. Dwyer, editor, *Model Checking Software, Proceedings of the 8th International SPIN Workshop (SPIN 2001), Toronto, Canada*, Lecture Notes in Computer Science, pages 123–142. Springer-Verlag, 2001.
14. M. S. Hecht. *Flow Analysis of Programs*. North-Holland, 1977.
15. G. Holzmann and J. Patti. Validating SDL specifications: an experiment. In E. Brinksma, editor, *International Workshop on Protocol Specification, Testing and Verification IX (Twente, The Netherlands)*, pages 317–326. North-Holland, 1989. IFIP TC-6 International Workshop.
16. G. J. Holzmann. *The Spin Model Checker*. Addison-Wesley, 2003.
17. N. Ioustinova, N. Sidorova, and M. Steffen. Abstraction and flow analysis for model checking open asynchronous systems. In *Proceedings of the 9th Asia-Pacific Software Engineering Conference (APSEC 2002, 4.–6. December 2002, Gold Coast, Queensland, Australia*, pages 227–235. IEEE Computer Society, Dec. 2002.
18. N. Ioustinova, N. Sidorova, and M. Steffen. Closing open SDL-systems for model checking with DT Spin. In L.-H. Eriksson and P. A. Lindsay, editors, *Proceedings of Formal Methods Europe (FME'02)*, volume 2391 of *Lecture Notes in Computer Science*, pages 531–548. Springer-Verlag, 2002.
19. G. Kildall. A unified approach to global program optimization. In *Proceedings of POPL '73*, pages 194–206. ACM, January 1973.
20. O. Kupferman and M. Y. Vardi. Module checking revisited. In O. Grumberg, editor, *CAV '97, Proceedings of the 9th International Conference on Computer-Aided Verification, Haifa. Israel*, volume 1254 of *Lecture Notes in Computer Science*. Springer, June 1997.
21. O. Kupferman, M. Y. Vardi, and P. Wolper. Module checking. In R. Alur, editor, *Proceedings of CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 75–86, 1996.
22. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Twelfth Annual Symposium on Principles of Programming Languages (POPL) (New Orleans, LA)*, pages 97–107. ACM, January 1985.
23. D. Long. *Model Checking, Abstraction and Compositional Verification*. PhD thesis, Carnegie Mellon University, 1993.
24. F. Nielson, H.-R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
25. A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed, and J. R. W. Smith. *System Engineering Using SDL-92*. Elsevier Science, 1997.
26. A. Pnueli. The temporal logic of programs. In *Proceeding of the 18th Annual Symposium on Foundations of Computer Science*, pages 45–57, 1977.
27. Specification and Description Language SDL. CCITT, 1993.
28. Specification and Description Language SDL, blue book. CCITT Recommendation Z.100, 1992.
29. N. Sidorova and M. Steffen. Embedding chaos. In P. Cousot, editor, *Proceedings of SAS'01*, volume 2126 of *Lecture Notes in Computer Science*, pages 319–334. Springer-Verlag, 2001.

30. N. Sidorova and M. Steffen. Verifying large SDL-specifications using model checking. In R. Reed and J. Reed, editors, *Proceedings of the 10th International SDL Forum SDL 2001: Meeting UML*, volume 2078 of *Lecture Notes in Computer Science*, pages 403–416. Springer-Verlag, Feb. 2001.
31. N. Sidorova and M. Steffen. Synchronous closing of timed SDL systems for model checking. In A. Cortesi, editor, *Proceedings of the Third International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI) 2002*, volume 2294 of *Lecture Notes in Computer Science*, pages 79–93. Springer-Verlag, 2002.
32. A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1992. Earlier version in the proceeding of CAV '90 Lecture Notes in Computer Science 531, Springer-Verlag 1991, pp. 156–165 and in Computer-Aided Verification '90, DIMACS Series in Discrete Mathematics and Theoretical Computer Science Vol. 3, AMS & ACM 1991, pp. 25–41.
33. A wireless ATM network demonstrator (WAND), ACTS project AC085. `http://www.tik.ee.ethz.ch/~wand/`, 1998.