Synchronous Closing of Timed SDL Systems for Model Checking

February 20, 2002

Natalia Sidorova[1] and Martin Steffen[2]

[1] 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
[2] Institut für angewandte Mathematik und Informatik
Christian-Albrechts-Universität
Preußerstraße 1–9,
24105 Kiel, Deutschland
ms@informatik.uni-kiel.de

Abstract. Standard model checkers cannot handle open reactive systems directly. Closing the system is commonly done by adding an environmental process. However, for model checking, the way of closing should be well-considered to alleviate the state-space explosion problem. This is especially true in the context of model checking SDL with its asynchronous message-passing communication because of a combinatorial explosion caused by all combinations of messages in the input queues.

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.

1 Introduction

Model checking [7] 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 [11, 25] and abstraction [19, 7, 9].

As standard model checkers, e.g., Spin [14], 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. 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 SDL-programs (*Specification and Description Language*) [22] with its *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 [23] 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. 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. 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, since we are dealing with the discrete-time semantics [13, 3] of SDL, 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,* as in [23], we 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 [21, 18], we thus get the desired property preservation: if $S^\sharp \models \varphi$ then $S \models \varphi$.

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 over-approximate the behavior. In Section 5 we discuss future work.

2 Semantics

In this section, we fix syntax and semantics of our analysis. As we take SDL [22] as source language, our operational model is based on asynchronously com-

municating 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 $Chan_i$ and $Chan_o$, 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 $(P, (in, out), Var, Loc, \sigma_{init}, Edg)$, where (in, out) are the finite sets of *input* resp. *output* channel names of the process, *Var* denotes a finite set of variables, and *Loc* denotes a finite set of *locations* or control states. 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. For a process P_i in a parallel composition, we write $\bar{P}$ for its environment, i.e., all processes except P. A mapping from variables to values is called a valuation; we denote the set of valuations by $Val = Var \to D$. 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. $\Sigma = Loc \times Val$ is the set of states, where each process has one designated initial state $\sigma_{init} = (l_{init}, \eta_{init}) \in \Sigma$. 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.

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.* In SDL, each transition starts with an input action, hence 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 $c?s(x)$, $g \rhd c!s(e)$, and $g \rhd x := e$, respectively, and we use α, $\alpha' \ldots$ when leaving the class of actions unspecified. For an edge $(l, \alpha, \hat{l}) \in Edg$, we write more suggestively $l \longrightarrow_{\alpha} \hat{l}$. We assume for the non-timer guards, that at least one of them evaluates to true in each state. This assumption corresponds at the SDL source language level 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 case-construct.

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 $t, t'_1, \ldots$), where each timer variable consists of a boolean flag indicating whether the timer is active or not, together with a natural number value denoting its expiration time. A timer can be either *set* to a value, i.e., activated to run for the designated period, or *reset,* i.e., deactivated. Setting and resetting are expressed by guarded actions of the form $g \rhd set\ t := e$ and $g \rhd reset\ t$. 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 $g_t \rhd reset\ t$, 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 described by sequences of states $\sigma_{init} = \sigma_0 \to_{\lambda} \sigma_1 \to_{\lambda} \ldots$ starting from the initial one. The step semantics $\to_{\lambda} \subseteq \Sigma \times Lab \times \Sigma$ is given as a labeled transition relation between states. The labels

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

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

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

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

$$\frac{l \longrightarrow_{g \rhd reset\ t} \hat{l} \in Edg \qquad [\![g]\!]_\eta = true}{(l,\eta) \rightarrow_{\tau} (\hat{l}, \eta_{[t \mapsto off]})}\ \textsc{Reset}$$

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

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

$$\frac{blocked(\sigma)}{\sigma \rightarrow_{tick} \sigma_{[t \mapsto (t-1)]}}\ \textsc{Tick}_P$$

Table 1. Step semantics for process P

differentiate between internal τ-steps, "*tick*"-steps, which globally decrease all active timers, and communication steps, either input or output, which are labeled by a triple of channel name, signal, and transmitted value. Depending on location, valuation, and the potential next actions, the possible successor states are given by the rules of Table 1.

Inputting a value means reading a value belonging to a matching signal from the channel and updating the local valuation accordingly (rule INPUT), where $\eta \in Val$, and $\eta_{[x \mapsto v]}$ 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 of SDL-92 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. Unlike input, output is guarded, so sending a message involves evaluating the guard and the expression according to the current valuation (rule OUTPUT). Assignment in ASSIGN works analogously, except that the step is internal.

Concerning the temporal behavior, timers are treated in valuations as variables, distinguishing active and deactivated timer. The *set*-command activates a timer, setting its value to the specified time, *reset* deactivates it; both actions

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

$$\frac{\mathit{blocked}(c,q)}{(c,q) \rightarrow_{\mathit{tick}} (c,q)}\ \textsc{Tick}_Q$$

Table 2. Step semantics for a queue

are guarded (cf. rules SET and RESET). A timeout may occur, if an active timer has expired, i.e., reached zero (rule TIMEOUT).

Time elapses by counting down active timers till zero, which happens in case no untimed actions are possible. In rule TICK, this is expressed by the predicate *blocked* on states: $\mathit{blocked}(\sigma)$ holds if no move is possible except either a clock-tick or a reception of a message, i.e., if $\sigma \rightarrow_\lambda$ for some label λ, then $\lambda = \mathit{tick}$ 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., $\mathit{on}(n+1) - 1 = \mathit{on}(n)$, non-active timers are not affected. Note that the operation is undefined for $\mathit{on}(0)$, which is justified later by Lemma 1.

In SDL, timeouts are often considered as specific timeout *messages* kept in the input 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 modeled 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 chosen here is not the only one conceivable (see e.g. [5] for a broader discussion of the use of timers in SDL). The semantics we use is the one described in [13, 3], and is also implemented in DTSpin [2, 10], a discrete time extension of the Spin model checker.

In SDL's asynchronous communication model, a process receives messages via a single associated input queue. We write ϵ for the empty queue; $(s,v)::q$ denotes a queue with message (s,v) (consisting of a signal s and a value 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. To facilitate the comparison of the asynchronous with the synchronous behavior of the environment, 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. In abuse of notation and to allow a uniform presentation of parallel composition below, we use the symbol σ not only for typical element of process states, but also for states (c,q) of queues. We require for the input and the output channel names of a queue that $\mathit{in}(c) = \{c_o\}$ and $\mathit{out}(c) = \{c_i\}$. The operational rules for queues are shown in Table 2.

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

$$\frac{\gamma_1 \rightarrow_\tau \hat{\gamma}_1}{(\gamma_1, \gamma_2) \rightarrow_\tau (\hat{\gamma}_1, \gamma_2)} \textsc{Interleave}_\tau$$

$$\frac{\gamma_1 \rightarrow_{c?(s,v)} \hat{\gamma}_1 \qquad c \notin \mathit{out}(\gamma_2)}{(\gamma_1, \gamma_2) \rightarrow_{c?(s,v)} (\hat{\gamma}_1, \gamma_2)} \textsc{Interleave}_{in}$$

$$\frac{\gamma_1 \rightarrow_{c!(s,v)} \hat{\gamma}_1 \qquad c \notin \mathit{in}(\gamma_2)}{(\gamma_1, \gamma_2) \rightarrow_{c!(s,v)} (\hat{\gamma}_1, \gamma_2)} \textsc{Interleave}_{out}$$

$$\frac{\gamma_1 \rightarrow_{tick} \hat{\gamma}_1 \qquad \gamma_2 \rightarrow_{tick} \hat{\gamma}_2}{(\gamma_1, \gamma_2) \rightarrow_{tick} (\hat{\gamma}_1, \hat{\gamma}_2)} \textsc{Tick}$$

Table 3. Parallel composition of R_1 and R_2

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 exactly if it is empty. Note further 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.

The semantics for parallel composition of processes or queues is given by the rules of Table 3. We call the parallel composition of one or more local states (either of processes or queues) a *configuration* and write $\gamma, \gamma'_1 \ldots \in \Gamma$ for typical elements. This means, γ is a vector of states of the participating processes or queues. Since we assumed that the variable sets of the components are all disjoint, we write $\gamma(x)$ for the value $\eta(x)$, for one state $\sigma = (l, \eta)$ being part of γ; analogously, we use the notation $[\![e]\!]_\gamma$ for the value of e in γ. The *initial* configuration of a parallel composition of components is given by the array of initial process states together with empty queues. We call a sequence of configurations $\gamma_{init} = \gamma_0 \rightarrow_\lambda \gamma_1 \rightarrow_\lambda \ldots$ starting from the initial configuration γ_{init} a *run*.

Communication between two partners is done by exchanging a common signal s and value v over a channel name c, as given by rule COMM. Note that by our conventions, $c \in \mathit{out}(\sigma_1)$ as well as $c \in \mathit{in}(\sigma_2)$. Note further that by the syntactic restrictions on the use of input and output channel names, only synchronization between one process and a queue can happen. As far as τ-steps and non-matching communication messages are concerned, each process can proceed on its own by rule INTERLEAVE. Each rule has a symmetric counterpart, which we elide. Finally, two components can perform a tick-step if both are able to do so.

By connecting processes with queues, the above semantics describes *asynchronous* communication. Synchronous communication for a channel name c is characterized similarly by identifying the names c_o and c_i such that the two pro-

cesses *directly* communicate with each other. Furthermore, synchronous channels are not represented as queues in the system configuration.

Lemma 1. *Let S be a system and $\gamma \in \Gamma$ a configuration.*

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

Proof. If, for part (1), $[\![t]\!]_\eta = on(0)$ for a timer t in a process P, then either TIMEOUT or TDISCARD of Table 1 allow a τ-step for P. Hence, P is not *blocked* and therefore cannot do a *tick*-step. Consequently, the system cannot perform a *tick*-step. Part (2) follows from the fact that a queue 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.

Lemma 2. *For a configuration γ, $blocked(\gamma)$ iff $blocked(\sigma)$ for all states σ part of γ.*

3 Replacing asynchronous with synchronous communication

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 react to 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 changed the state of the process. Another effect of an input queue is that the queue introduces an arbitrary delay between reception of a message and the future reaction of the receiving process to this message. For an output, the effect is converse. This implies that 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.[1] This is related to the so-called *Brock-Ackerman anomaly*, characterizing the difference between buffered and unbuffered communication [6].

[1] 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.

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., at the steps between two ticks,[2] and not for 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 reduction sequence 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;*
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.*

Given a synchronous and an asynchronous versions of a process and two corresponding configurations $\gamma_s = \sigma_s$ and $\gamma_a = (\sigma_a, (c_i, q_i), (c_o^1, q_1), \ldots, (c_o^k, q_k))$. Then define $\trianglerighteq$ as $\gamma_a \trianglerighteq \gamma_s$, if $\sigma_a = \sigma_s$. 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 transitions, ignoring the τ-steps: so define $\Rightarrow_\lambda$ as $\rightarrow_\tau^* \rightarrow_\lambda \rightarrow_\tau^*$ when $\lambda \neq \tau$, and as $\rightarrow_\tau^*$ else. Correspondingly, $\vec{\lambda}$ denotes a sequence of weak steps with labels from the sequence $\vec{\lambda}$.

Lemma 4. *Assume a synchronous and an asynchronous version P_s and P_a of a process and corresponding configurations 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 reduction sequence, where $\vec{\lambda}$ does not contain a tick-step, 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 i
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 INPUT where the process does a step $\sigma \rightarrow_{c?(s,v)} \sigma'$ and the queue the corresponding output step by rule OUT, are replaced by a

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

direct input step $\gamma_s \rightarrow_{c?(s,v)} \gamma'_s$. 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
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.

Theorem 5. *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 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 following definition of simulation, which considers as basic steps only tick-steps or else the sequence of steps within one time slice. A binary relation $R \subseteq \Gamma_1 \times \Gamma_2$ on two sets of configurations is called a *tick*-simulation, when the following conditions hold:

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

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

We define the relation $R \subseteq \Gamma_a \times \Gamma_s$ as $(l_s, \eta_s, ((c_i, q_0), (c_o^1, q_1), \ldots, (c_o^k, q_k))) \; R \; (l_s, \eta_s)$ iff $(l_s, \eta_s) = (l_a, \eta_a)$ and $q_i = \epsilon$ for all queues. To show that R is indeed a tick-simulation, assume $\gamma_a = (l, \eta, ((c_i, \epsilon), (c_o^1, \epsilon), \ldots, (c_o^k, \epsilon)))$ and $\gamma_s = (l, \eta)$ with $\gamma_a \; R \; \gamma_s$. 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, $blocked(\gamma_a)$ must hold, i.e., there are no steps enabled except input from the outside or *tick*-steps. Since immediately $blocked(\gamma_s)$, also $\gamma_s \rightarrow_{tick} \gamma_{s[t \mapsto (t-1)]}$, which concludes the case.

Case: $\gamma_a \Rightarrow_{\vec{\lambda}} \gamma'_a$
where $blocked(\gamma'_a)$ 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 configurations are in relation R as defined above, this gives $P_a \preceq_{tick} P_s$. It can be shown by a standard argument, that this implies $[\![P_a]\!]_{wtrace} \subseteq [\![P_s]\!]_{wtrace}$, as required. □

4 Abstracting data

In this section, we present a straightforward dataflow analysis marking variable and timer instances that may be influenced by the environment. It is a minor adaptation of the one from [23], taking care of channel communication.

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 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.

The analysis works on an abstract representation of the data values, where $\top$ is interpreted as value chaotically influenced by the environment and $\bot$ stands for a non-chaotic value. We write $\eta^\alpha, \eta^\alpha_1, \ldots$ for abstract valuations, i.e., for typical elements from $Val^\alpha = Var \rightarrow \{\top, \bot\}$. 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 $\eta^\alpha_1, \ldots, \eta^\alpha_n$ by $\bigvee_{i=1}^n \eta^\alpha_i$ (or by $\eta^\alpha_1 \vee \eta^\alpha_2$ in the binary case).

Each node n of the flow graph has associated an abstract transfer function $f_n : Val^\alpha \rightarrow Val^\alpha$, as given in Table 4, where α_n denotes the action associated with the node n of process P. The equations are mostly straightforward, describing the change the abstract valuations depending on the sort of action at the node. The only case deserving mention is the one for $c_i?s(x)$, whose equation captures the inter-process data-flow from a sending to a receiving actions (using c_i and c_o, we assume asynchronous communication in the analysis). In the equation $\bar{P}$ stands for the environment of P, i.e., the rest of the system. It is easy to see that the functions f_n are monotone.

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: $\eta^\alpha_{init}(n) = \eta_\bot$. This choice rests on the assumption that all local variables of each process are properly

$$
\begin{array}{rll}
f(c_i?s(x))\eta^\alpha & = \begin{cases} \eta^\alpha[x \mapsto \top] & c \notin out(\bar{P}) \\ \eta^\alpha[x \mapsto \bigvee\{[\![e]\!]_{\eta^\alpha} \mid \alpha_{n'} = g \rhd c_o!s(e) \text{ for some node } n'\}] & \text{else} \end{cases} \\
f(g \rhd c_o!s(e))\eta^\alpha & = \eta^\alpha & \\
f(g \rhd x := e)\eta^\alpha & = \eta^\alpha[x \mapsto [\![e]\!]_{\eta^\alpha}] & \\
f(g \rhd set\ t := e)\eta^\alpha & = \eta^\alpha[t \mapsto on([\![e]\!]_{\eta^\alpha})] & \\
f(g \rhd reset\ t)\eta^\alpha & = \eta^\alpha[t \mapsto off] & \\
f(g_t \rhd reset\ t)\eta^\alpha & = \eta^\alpha[t \mapsto off] &
\end{array}
$$

Table 4. Transfer functions/abstract effect for process P

initialized. We are interested in the least solution to the data-flow problem given by the following constraint set:

$$
\begin{array}{l}
\eta^\alpha_{post}(n) \geq f_n(\eta^\alpha_{pre}(n)) \\
\eta^\alpha_{pre}(n) \geq \bigvee\{\eta^\alpha_{post}(n') \mid (n', n) \text{ in flow relation}\}
\end{array} \tag{1}
$$

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 4. The least fixpoint of the constraint set can be solved iteratively in a fairly standard way by a *worklist algorithm* (see e.g., [15, 12, 20]), where the worklist steers the iterative loop until the least fixpoint is reached (cf. Fig. 1).

---

input : the flow-graph of the program
output: $\eta^\alpha_{pre}, \eta^\alpha_{post}$;

$\eta^\alpha(n) = \eta^\alpha_{init}(n)$;
$WL = \{n \mid \alpha_n = ?s(x), s \in Sig_{ext}\}$;

repeat
  pick $n \in WL$;
  let $S = \{n' \in succ(n) \mid f_n(\eta^\alpha(n) \not\leq \eta^\alpha(n')\}$
  in
    for all $n' \in S$: $\eta^\alpha(n') := f(\eta^\alpha(n))$;
    $WL := WL \backslash n \cup S$;
until $WL = \emptyset$;

$\eta^\alpha_{pre}(n) = \eta^\alpha(n)$;
$\eta^\alpha_{post}(n) = f_n(\eta^\alpha(n))$

Fig. 1. 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, and 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 [20] 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 6 (Correctness). *Upon termination, the the algorithm gives back the least solution to the constraint set as given by the equations (1), resp. Table 4.*

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 transformation given as a set of transformation rules for each process P, similar to the ones from [23]. As the transformation here is simpler (since it does not embed the environment process $\bar{P}$ by incorporating its effect directly into P) we omit the full set of rules. The transformation is straightforward: guards potentially influenced by the environment are taken non-deterministically, i.e., a guard g at a location l is replaced by *true*, if $[\![g]\!]_{\eta^{\alpha}_{l}} = \top$. Assignments of expressions whose value may depend on data from the environment are omitted. For timer guards whose value is indeterminate because of outside influence, we work with a 3-valued abstraction: *off* when the timer is deactivated, a value $on(\top)$ when the timer is active with arbitrary expiration time, and a value $on(\top^{+})$ for active timers, whose expiration time is arbitrary except immediate timeout; the latter two abstract values are represented by $on(0)$ and $on(1)$, respectively, and the non-deterministic behavior of the timer expiration is captured by arbitrarily postponing a timeout by setting back the value of the timer to $on(1)$. This is captured by adding edges according to:

$$\frac{[\![t]\!]_{\eta_l^\alpha} = \top}{l \longrightarrow_{g_t \,\triangleright\, reset\ t} \longrightarrow_{set\ t:=1} l \in Edg^\sharp} \text{T-NoTimeout}$$

As the transformation only adds non-determinism, the transformed system $S^\sharp$ simulates S (cf. [23]). Together with Theorem 5, 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 influence data is not much of an additional loss of precision.

Lemma 7. *Let P_a and P_s be the asynchronous resp. synchronous variant of a process, and S be given as the parallel composition of a $P_a \parallel \bar{P}$, where $\bar{P}$ is the environment of P. Furthermore, let $S^\sharp = P_s^\sharp \parallel \bar{P}$ be defined as before, and φ a next-free LTL-formula mentioning only variables from $\{x \mid \neg\exists l \in Loc.\ [\![x]\!]_{\eta_l^\alpha} = \top\}$. Then $S^\sharp \models \varphi$ implies $S \models \varphi$.*

5 Conclusion

In this paper, we extended earlier work from [23] describing how to close an open, asynchronous SDL-process by a timed chaotic 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 and separating, more or less, input and output.

In the context of software-testing, [8] 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 [4]

A fundamental approach to model checking open systems is known as *module* checking [17][16]. 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 [24] using the chaotic closure to this more general setting. In the experiments, we are using a JAVA-implementation of the automatic closing and the dataflow algorithm for concrete SDL-92 resp. a discrete-time extension of the Spin model checker which we use in the verification. 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, 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. Discrete-time Spin. `http://win.tue.nl/~dragan/DTSpin.html`, 2000.
11. 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.
12. M. S. Hecht. *Flow Analysis of Programs*. North-Holland, 1977.
13. 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.

14. G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.
15. G. Kildall. A unified approach to global program optimization. In *Proceedings of POPL '73*, pages 194–206. ACM, January 1973.
16. 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.
17. 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.
18. 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.
19. D. Long. *Model Checking, Abstraction and Compositional Verification.* PhD thesis, Carnegie Mellon University, 1993.
20. F. Nielson, H.-R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.
21. A. Pnueli. The temporal logic of programs. In *Proceeding of the 18th Annual Symposium on Foundations of Computer Science*, pages 45–57, 1977.
22. Specification and Description Language SDL, blue book. CCITT Recommendation Z.100, 1992.
23. N. Sidorova and M. Steffen. Embedding chaos. In P. Cousot, editor, *Proceedings of the 8th Static Analysis Symposium (SAS'01)*, volume 2126 of *Lecture Notes in Computer Science*, pages 319–334. Springer-Verlag, 2001.
24. 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.
25. 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.