Embedding Chaos

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. Model checking would answer all finite-state verification problems, if it were not for the notorious state-space explosion problem. A problem of practical importance, which attracted less attention, is to *close* open systems. Standard model checkers cannot handle open systems directly and closing is commonly done by adding an environment process, which in the simplest case behaves *chaotically*. 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, since chaotically sending and receiving messages immediately leads to a combinatorial explosion caused by all combinations of messages in the input queues.

In this paper we develop an automatic transformation yielding a closed system. By *embedding* the outside chaos into the system's processes, we avoid the state-space penalty in the input queues mentioned above. To capture the chaotic timing behaviour of the environment, we introduce a non-standard 3-valued timer abstraction. We use *data-flow analysis* to detect instances of chaotic variables and timers and prove the soundness of the transformation, which is based on the result of the analysis.

Keywords: model checking, open reactive systems, data-flow analysis, SDL

1 Introduction

Model checking [10] is considered as method of choice in the verification of reactive systems and is increasingly accepted in industry for its push-button appeal. To alleviate the notorious state-space explosion problem, a host of techniques has been invented, e.g., partial-order reduction [19, 37] and abstraction [28, 10, 13], to mention two prominent approaches.

A problem of practical importance, which attracted less attention, is to *close* open systems. Since standard model checkers, e.g., Spin [23], cannot handle open systems, one first has to transform the model into a closed one. This is commonly done by adding an environment process that, in order to be able to infer properties for the concrete system, must exhibit at least all the behaviour of the real environment. The simplest safe abstraction of the environment thus behaves *chaotically.* When done manually, this closing, as simple as it is, is tiresome and error-prone for large systems already due to the sheer amount of signals. Moreover, for model checking, 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*) [33] with its *asynchronous* message-passing communication model. Sending arbitrary message streams to the unbounded input queues will immediately lead to an infinite state space, unless some assumptions restricting the environment behaviour are incorporated in the closing process. Even so, external chaos results in a combinatorial explosion caused by all combinations of messages in the input queues. This way of closing is even more wasteful, since most of the messages are dropped by the receiver due to the discard-feature of SDL-92.

Another problem the closing must address is that the *data* carried with the messages coming from the environment is usually drawn from some infinite data domain. Since furthermore we are dealing with the discrete-time semantics [22, 7] of SDL, special care must be taken to ensure that the chaos also shows more behaviour wrt. *timing* issues such as timeouts and time progress.

To solve these three problems, we develop an automatic transformation yielding a closed system. (1) By *embedding* the outside chaos into the system's processes, we avoid the state-space penalty in the input queues mentioned above. (2) We use *data abstraction,* condensing data from outside into a single abstract value $\mathbb{T}$ to deal with the infinity of environmental data. In effect, by embedding the chaos process and abstracting the data, there is no need to ever consider messages from the outside at all. Hence, the transformation removes the corresponding input statements. By removing reception of chaotic data, we nevertheless must take into account the cone of influence of the removed statements, lest we get less behaviour than before. Therefore, we use *data-flow analysis* to detect instances of chaotically influenced variables and timers. (3) To capture the chaotic timing behaviour, we introduce a non-standard 3-valued timer abstraction.

Based on the result of the analysis, the transformation yields a *closed* system $S^\sharp$ which shows more behaviour in terms of traces than the original one. For formulas of next-free LTL [32, 27], we thus get the desired property preservation: if $S^\sharp \models \varphi$ then $S \models \varphi$.

The remainder of the paper is organized as follows. Section 2 introduces syntax and semantics we use, modelling the communication and timed behaviour of SDL. In Section 3 we present the data-flow algorithm marking variable and timer instances influenced by chaos. Section 4 then develops the transformation and proves its soundness. Finally in Section 5 we conclude with related and future work.

2 Semantics

In this section, we fix syntax and semantics of our analysis. Since we take SDL [33] as source language, our operational model is based on asynchronously communicating state machines (processes) with top-level concurrency. 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 four-tuple $(Var, Loc, \sigma_{init}, Edg)$, where 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. A mapping of 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., and write D when leaving the data-domain unspecified, and silently assume all expressions to be well-typed. $\Sigma = Loc \times Val$ is the set of states, where a process has one designated initial state $\sigma_{init} = (l_{init}, Val_{init}) \in \Sigma$. An *edge* of the state machine describes a change of configuration resulting from performing an *action* from a set Act of actions; the set $Edg \subseteq Loc \times Act \times Loc$ denotes the set of edges.

As actions, we distinguish (1) *input* of a signal s containing a value to be assigned to a local variable, (2) *sending* a signal s together with a value described by an expression to a process P', and (3) *assignments*. In SDL, each transition starts with an input action, hence we assume the inputs to be unguarded, while output and assignment can be *guarded* by a boolean expression g, its guard. The three classes of actions are written as $?s(x)$, $g \triangleright P!s(e)$, and $g \triangleright x := e$, respectively, and we use $\alpha, \alpha' \ldots$ when leaving the class of actions unspecified. For an edge $(l, \alpha, \hat{l}) \in Edg$, we write more suggestively $l \longrightarrow_\alpha \hat{l}$.

Time aspects of a system behaviour are specified by actions dealing with *timers*. Each process has a finite set of timer variables (with typical elements $t, t'_1, \ldots$) which consist of a boolean flag indicating whether the timer is active or not, and a natural number value. 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 $g \triangleright set\ t := e$ 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.

As the syntax of a program is given in two levels — state machines and their parallel composition — so is their *semantics.* In SDL's asynchronous communication model, a process receives messages via a single associated input queue. We call a state of a process together with its input queue a *configuration* (σ, q). 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. The behaviour of a single process is then given by sequences of configurations $(\sigma_{init}, \epsilon) = (\sigma_0, q_0) \rightarrow_\lambda (\sigma_1, q_1) \rightarrow_\lambda \ldots$ starting from the initial one, i.e., the initial state and the empty queue. The step semantics $\rightarrow_\lambda \subseteq \Gamma \times Lab \times \Gamma$ is given as a labelled transition relation between configurations. The labels differentiate between internal τ-steps, "$tick$"-steps, which globally decrease all active timers, and communication steps, either input or output, which are labelled

$$\frac{l \longrightarrow_{?s(x)} \hat{l} \in Edg}{(l, \eta, (s, v) :: q) \rightarrow_\tau (\hat{l}, \eta_{[x \mapsto v]}, q)} \text{ INPUT} \qquad \frac{l \longrightarrow_{?s'(x)} \hat{l} \in Edg \Rightarrow s' \neq s}{(l, \eta, (s, \_) :: q) \rightarrow_\tau (l, \eta, q)} \text{ DISCARD}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, P'!(s,e)} \hat{l} \in Edg \qquad [\![g]\!]_\eta = true \qquad [\![e]\!]_\eta = v}{(l, \eta, q) \rightarrow_{P'!(s,v)} (\hat{l}, \eta, q)} \text{ OUTPUT}$$

$$\frac{v \in D}{(l, \eta, q) \rightarrow_{P?(s,v)} (l, \eta, q :: (s, v))} \text{ RECEIVE}$$

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

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

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

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

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

Table 1. Step semantics for process P

by a triple of process (of destination/origin resp.), signal, and value being transmitted. Depending on location, valuation, the possible next actions, and the content of the input queue, the possible successor configurations are given by the rules of Table 1, where we assume a given set Sig_{ext} of signals exchanged with the environment.

Inputting a value means reading a value belonging to a matching signal from the head of the queue and updating the local valuation accordingly (rule INPUT), where $\eta \in Val$, and $\eta_{[x \mapsto v]}$ stands for the valuation equalling 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 head of the input queue cannot be reacted upon at the current control state, i.e., 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. Receiving a message by asynchronous communication simply means putting it into the input queue where in the RECEIVE-rule, P is the identity of the process.

Concerning the temporal behaviour, timers are treated in valuations as variables, distinguishing active and deactivated timer. The set-command activates a timer, setting its value to a specified time, $reset$ deactivates it; both actions are guarded (cf. rules SET

$$\frac{(\sigma_1, q_1) \rightarrow_{P!(s,v)} (\hat{\sigma}_1, \hat{q}_1) \qquad (\sigma_2, q_2) \rightarrow_{P?(s,v)} (\hat{\sigma}_2, \hat{q}_2) \qquad s \notin Sig_{ext}}{(\sigma_1, q_1) \times (\sigma_2, q_2) \rightarrow_{\tau} (\hat{\sigma}_1, \hat{q}_1) \times (\hat{\sigma}_2, \hat{q}_2)} \ \textsc{Comm}$$

$$\frac{(\sigma_1, q_1) \rightarrow_{\lambda} (\hat{\sigma}_1, \hat{q}_1) \qquad \lambda = \{\tau, P?(s,v), P!(s,v) \mid s \in Sig_{ext}\}}{(\sigma_1, q_1) \times (\sigma_2, q_2) \rightarrow_{\lambda} (\hat{\sigma}_1, \hat{q}_1) \times (\sigma_2, q_2)} \ \textsc{Interleave}$$

$$\frac{blocked(l, \eta, q)}{(l, \eta, q) \rightarrow_{tick} (l, \eta_{[t \mapsto (t-1)]}, q)} \ \textsc{Tick}$$

Table 2. Parallel composition

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

We assume for the non-timer-guards, that at least one of them evaluates to true for each configuration.[1]

The *global* transition semantics for a program $Prog = \Pi_{i=1}^{n} P_i$ is given by a standard product construction: configurations and initial states are paired, and global transitions synchronize via their common labels. The global step relation $\rightarrow_\lambda \subseteq \Gamma \times Lab \times \Gamma$ is given by the rules of Table 2.

Asynchronous communication between the two processes uses a system-internal signal s to exchange a common value v, as given by rule COMM. As far as τ-steps and communication messages using external signals are concerned, each process can proceed on its own by rule INTERLEAVE. Both rules have a symmetric counterpart, which we elide. 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 configurations: $blocked(\gamma)$ holds if no move is possible by the system except either a clock-tick or a reception of a message from the outside, i.e., if $\gamma \rightarrow_\lambda$ for some label λ, then $\lambda = tick$ or $\lambda = P?(s,v)$ for some $s \in Sig_{ext}$. In other words, the time-elapsing steps are those with *least priority.* Note in passing that due to the discarding feature, $blocked(\sigma, q)$ implies $q = \epsilon$. 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 by the following lemma.

Lemma 1. *Let S be a system and $(l, \eta, q) \in \Gamma$ a configuration. If $(l, \eta, q) \rightarrow_{tick} (l, \eta', q)$, then $[\![t]\!]_\eta \neq on(0)$, for all timers t.*

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 modelled more directly by

[1] 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.

guards. The equivalence of timeouts-by-guards and timeouts-as-messages in the presence of SDL's asynchronous communication model is argued for in [7]. The semantics we use is the one described in [22, 7], and is also implemented in *DTSpin* [6, 15], a discrete time extension of the *Spin* model checker.

3 Marking chaotically-influenced variable and timer instances

In this section, we present a straightforward dataflow analysis marking variable and timer instances that may be influenced by the chaotic environment. The analysis forms the basis of the transformation in Section 4.

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 $\eta^\alpha, \eta_1^\alpha, \ldots$ 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).

Each node n of the flow graph has associated an abstract transfer function $f_n : Val^\alpha \to Val^\alpha$. The functions are given in Table 3, where α_n denotes the action associated with the node n. 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 $?s(x)$, whose equation captures the inter-process data-flow from a sending to a receiving actions. It is easy to see that the functions f_n are monotone.

$$
\begin{aligned}
f(?s(x))\eta^\alpha &= \begin{cases} \eta^\alpha[x \mapsto \top] & s \in Sig_{ext} \\ \eta^\alpha[x \mapsto \bigvee\{\llbracket e \rrbracket_{\eta^\alpha} \mid \alpha_{n'} = g \rhd P!s(e) \text{ for some node } n'] & \text{else} \end{cases} \\
f(g \rhd P!s(e))\eta^\alpha &= \eta^\alpha \\
f(g \rhd x := e)\eta^\alpha &= \eta^\alpha[x \mapsto \llbracket e \rrbracket_{\eta^\alpha}] \\
f(g \rhd set\ t := e)\eta^\alpha &= \eta^\alpha[t \mapsto on(\llbracket e \rrbracket_{\eta^\alpha})] \\
f(g \rhd reset\ t)\eta^\alpha &= \eta^\alpha[t \mapsto \mathit{off}] \\
f(g_t \rhd reset\ t)\eta^\alpha &= \eta^\alpha[t \mapsto \mathit{off}]
\end{aligned}
$$

Table 3. Transfer functions/abstract effect for process P

Upon start of the data-flow 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:

$$\eta^{\alpha}_{post}(n) \geq f_n(\eta^{\alpha}_{pre}(n)) \tag{1.1}$$

$$\eta^{\alpha}_{pre}(n) \geq \bigvee\{\eta^{\alpha}_{post}(n') \mid (n', n) \text{ in flow relation}\} \tag{1.2}$$

For each node n of the flow graph, the data-flow problem is specified by two in-equations 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 3. The least fixpoint of the constraint set can be solved iteratively in a fairly standard way by a *worklist algorithm* (see e.g. [24, 21, 30]), where the worklist steers the iterative loop until the least fixpoint is reached. The algorithm for our problem is shown in 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.

Lemma 2 (Termination). *The algorithm of Figure 1 terminates.*

Proof. Immediate, for set of variables Var in the program is finite and hence the lattice $Val^{\alpha} = Var \to \{\top, \bot\}$ is finite, as well. □

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 [30] for a broader general 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}. The definition is justified by the following observation:

Lemma 3. *Given a location l and a node $\hat{n}$ from the flow graph such that $\hat{n} = l \longrightarrow_{\hat{\alpha}} \hat{l}$. Then $\eta^{\alpha}_{pre}(\hat{n}) = \bigvee\{\eta^{\alpha}_{post}(\tilde{n}) \mid \tilde{n} = \tilde{l} \longrightarrow_{\alpha} l\}$.*

4 Closing the system

The analysis marks instances of variables and timers potentially influenced by the chaotic environment. Based on this information, we transform the given system into a closed one, which shows more behaviour than the original. Since for model checking, we cannot live with the infinity of data injected from outside by the chaotic environment, we abstract this infinity into one single abstract value $\mathbb{T}$. For chaotically influenced timer values, we will need a more refined abstraction using 3 different values (cf. Section 4.1). Since the abstract system is still open, we close it in a second step, also implementing the abstract values by concrete ones (cf. Section 4.2). With the chaotic environment embedded into the now closed system, we remove, as optimization for model checking, external signals from the input queues. Special care is taken to properly embed the chaotic behaviour wrt. the timed behaviour.

4.1 Abstracting data

As mentioned, we extend the data domains each by an additional value $\mathbb{T}$, representing data received from the outside, i.e., we assume now domains such as $\mathbb{N}^{\mathbb{T}} = \mathbb{N} \,\dot{\cup}\, \{\mathbb{T}\}$, $Bool^{\mathbb{T}} = Bool \,\dot{\cup}\, \{\mathbb{T}\}, \dots$, 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 $[\![.]\!]_{\eta^{\mathbb{T}}}$, i.e., $[\![e]\!]_{\eta^{\mathbb{T}}} = \top$, if e contains a variable x such that $\eta^{\mathbb{T}}(x) = \top$.

The step semantics is given (as before) by the rules of Tables 1 and 2, except the following differences. $\mathbb{T}$-valued guards behave as evaluating to $true$, i.e., they are replaced by $([\![g]\!]_{\eta^{\mathbb{T}}} = true) \vee ([\![g]\!]_{\eta^{\mathbb{T}}} = \mathbb{T})$. For the TIMEOUT- and the TDISCARD-rule, the premise concerning the timer remains $[\![t]\!]_{\eta^{\mathbb{T}}} = on(0)$. The RECEIVE-rule is

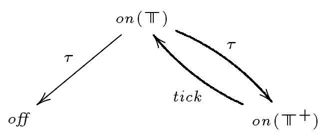

Fig. 2. Timer abstraction

replaced by I-ReceiveInt and I-ReceiveExt for internal and external reception, where the first one equals the old Receive when $s \notin Sig_{ext}$, and the latter postulates $(l, \eta, q) \rightarrow_{P?(s,\mathbb{T})} (l, \eta, q :: (s, \mathbb{T}))$ when $s \in Sig_{ext}$. To distinguish notationally the original system and its constituents from the intermediate one of this section, we write $P^{\mathbb{T}}$ for an intermediate-level process, $S^{\mathbb{T}}$ for an intermediate level system, etc.

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 in the on-state. Therefore, the domain of timer values contains as additional chaotic value $on(\mathbb{T})$. Since we need the intermediate system to show at least the behaviour of the original one, we must provide proper treatment of the rules involving $on(\mathbb{T})$, i.e., the Timeout-, the TDiscard-, and the Tick-rule. As $on(\mathbb{T})$ stands for any value of active timers, it must cover the cases where timeouts and timer-discards are enabled (because of $on(0)$) as well as disabled (because of $on(n)$ 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 $on(\mathbb{T}^+)$ for chaotic timers, representing all on-settings larger or equal 1. The non-deterministic choice between the two alternatives — zero and non-zero — is captured by the rules of Table 4. The order on the domain of timer values is given as smallest reflexive order relation such that $on(0) \leq on(\mathbb{T})$ and $on(n) \leq on(\mathbb{T}^+) \leq on(\mathbb{T})$, for all $n \geq 1$. The decreasing operation needed in the Tick-rule is defined in extension to the definition

$$\frac{[\![t]\!]_{\eta^{\mathbb{T}}} = on(\mathbb{T})}{(l, \eta^{\mathbb{T}}, q^{\mathbb{T}}) \rightarrow_\tau (l, \eta^{\mathbb{T}}[t \mapsto on(\mathbb{T}^+)], q^{\mathbb{T}})} \text{ I-NonZero}$$

$$\frac{l \longrightarrow_{g_t \,\triangleright\, reset\ t} \hat{l} \in Edg \qquad [\![t]\!]_{\eta^{\mathbb{T}}} = on(\mathbb{T})}{(l, \eta^{\mathbb{T}}, q) \rightarrow_\tau (l, \eta^{\mathbb{T}}[t \mapsto o\!f\!f], q)} \text{ I-Zero-Timeout}$$

$$\frac{(l \longrightarrow_\alpha \hat{l} \in Edg \Rightarrow \alpha \neq g_t \triangleright reset\ t) \qquad [\![t]\!]_{\eta^{\mathbb{T}}} = on(\mathbb{T})}{(l, \eta^{\mathbb{T}}, q^{\mathbb{T}}) \rightarrow_\tau (l, \eta^{\mathbb{T}}[t \mapsto o\!f\!f], q^{\mathbb{T}})} \text{ I-Zero-TDiscard}$$

Table 4. Non-determinism for $on(\mathbb{T})$

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

The intermediate system allows to state the soundness of the analysis: whenever a variable at some location contains a $\mathbb{T}$-value, the analysis has marked it by $\top$.

Theorem 5 (Soundness). *Given a system S, its intermediate representation $S^{\mathbb{T}}$, and η^α as the result of the analysis. Assume $\gamma_0^{\mathbb{T}} \to^* \gamma^{\mathbb{T}} = (l, \eta^{\mathbb{T}}, q^{\mathbb{T}})$, where $\gamma_0^{\mathbb{T}}$ is the initial configuration of $S^{\mathbb{T}}$. Then $[\![x]\!]_{\eta^{\mathbb{T}}} = \mathbb{T}$ implies $[\![x]\!]_{\eta_l^\alpha} = \top$, and $[\![t]\!]_{\eta^{\mathbb{T}}} \in \{on(\mathbb{T}), on(\mathbb{T}^+)\}$ implies $[\![t]\!]_{\eta_l^\alpha} = \top$.*

Next we make explicit the notion of simulation we will use to prove soundness of the abstraction. The new rule I-NONZERO introduces additional τ-steps in the intermediate system not present in the original behaviour. Hence, the simulation definition must honour additional τ-steps of $S^{\mathbb{T}}$ preceding a $tick$-step.

Definition 6 (Simulation). *Given two processes P and P' with sets of configurations Γ and Γ'. Assume further a relation $\leq \subseteq \Gamma \times \Gamma'$ on configurations and a relation $\leq \subseteq Lab \times Lab'$ on labels, denoted by the same symbol. A relation $R \subseteq \Gamma \times \Gamma'$ is a* simulation *if $R \subseteq \leq$, and if ($\gamma \; R \; \gamma'$ and $\gamma \to_\lambda \hat{\gamma}$) implies one of the following conditions:*

1. *If $\lambda \neq tick$, then $\gamma' \to_{\lambda'} \hat{\gamma}'$ and $\hat{\gamma} \; R \; \hat{\gamma}'$, for some configuration $\hat{\gamma}'$ and for some label $\lambda' \geq \lambda$.*
2. *If $\lambda = \tau$, then $\hat{\gamma} \; R \; \gamma'$.*
3. *If $\lambda = tick$, then $\gamma' = \gamma_0' \to_\tau \gamma_1' \to_\tau \ldots \to_\tau \gamma_n' \to_{tick} \hat{\gamma}'$ for some $n \geq 0$ such that $\hat{\gamma} \; R \; \hat{\gamma}'$ and $\gamma \; R \; \gamma_i'$ for all γ_i'.*

We write $P \preceq P'$, if there exists a simulation relation R such that $\gamma_{init} \; R \; \gamma'_{init}$ for the initial configurations γ_{init} and γ'_{init} of P resp. P'. The definition of simulation is analogously used for systems.

The simulation definition is given relative to order relations on configurations and on labels. To establish simulation concretely between S and $S^{\mathbb{T}}$, we define (in abuse of notation) for labels $\leq \subseteq Lab \times Lab^{\mathbb{T}}$ as the smallest relation such that $\tau \leq \tau$, and that $v \leq v^{\mathbb{T}}$ implies $P?(s, v) \leq P?(s, v^{\mathbb{T}})$ as well as $P!(s, v) \leq P!(s, v^{\mathbb{T}})$. We use the same symbol for the pointwise extension of $\leq$ to compare valuations, states, pairs $(s, v) \leq (s, v^{\mathbb{T}})$, queues, and finally configurations.

Lemma 7. *Let S and $S^{\mathbb{T}}$ as well as $\leq$ be defined as above. Then $S \preceq S^{\mathbb{T}}$.*

Proof sketch. It is straightforward to check on the rules of Table 1 that for single processes $P \preceq P^{\mathbb{T}}$. For systems, prove the implication that $P_1 \preceq P_1^{\mathbb{T}}$ and $P_2 \preceq P_2^{\mathbb{T}}$ implies $P_1 \parallel P_2 \preceq P_1^{\mathbb{T}} \parallel P_2^{\mathbb{T}}$, proceeding similarly by case analysis on the rules of Table 2. There, for the case of TICK, use the fact that $\gamma \leq \gamma^{\mathbb{T}}$ and $blocked(\gamma)$ implies $\gamma^{\mathbb{T}} = \gamma_0^{\mathbb{T}} \to_\tau \gamma_1^{\mathbb{T}} \to_\tau \ldots \to_\tau \gamma_n^{\mathbb{T}} = \hat{\gamma}^{\mathbb{T}}$ with $blocked(\hat{\gamma}^{\mathbb{T}})$ for some $n \geq 0$ and $\hat{\gamma}^{\mathbb{T}}$, and where furthermore $\gamma \leq \gamma_i^{\mathbb{T}}$ for all $\gamma_i^{\mathbb{T}}$. $\square$

Lemma 8. *Let systems S and $S^{\mathbb{T}}$ and the relations $\leq$ be defined as above. Then for all formulas φ from next-free LTL, $S \preceq S^{\mathbb{T}}$ and $S^{\mathbb{T}} \models \varphi$ implies $S \models \varphi$.*

4.2 Transformation

Based on the result of the analysis, we transform the given system S into an optimized one — we denote it by $S^\sharp$ — which is closed, which does not use the value $\mathbb{T}$, and which is in simulation relation with the original system.

In first approximation, the idea of the transformation is simple: just eliminate actions whose effect, judging from the results of the analysis, cannot be relied on. The transformation is given for each of the syntactic constructs by the rules of Table 5, where we denote a do-nothing statement by $skip$. The set of variables $Var^\sharp$ for $S^\sharp$ equals the original Var, except that for each process P of the system, a fresh timer-variable t_P is added to its local variables, i.e., $Var^\sharp_P = Var_P \dot\cup \{t_P\}$.

The transformation rules *embed* the chaotic environment's behaviour into a system. We start with the part not interacting with the environment, i.e., the transformation concerning the manipulation of variables and timers. Variable assignments are either left untouched or replaced by $skip$, depending on the result of the analysis concerning the left-hand value of the assignement (rules T-ASSIGN$_1$ and T-ASSIGN$_2$). A non-timer guard g at a location l is replaced by $true$, if $[\![g]\!]_{\eta^\alpha_l} = \top$; if not, the guard stays unchanged for the transformed system. We use $g^\sharp = [\![g]\!]_{\eta^\alpha_l}$ as shorthand for this replacement in the rules. For chaotic timers, we represent the abstract values $on(\mathbb{T})$ and $on(\mathbb{T}^+)$ of the intermediate system by the concrete $on(0)$ and $on(1)$, respectively, and directly incorporate the I-NONZERO-step of Table 4 by the transformation rule T-NOTIMEOUT.

For communication statements, we distinguish between signals going to or coming from the environment, and those exchanged within the system. Output to the outside basically is skipped (cf. rules T-OUTPUT$_1$ and T-OUTPUT$_2$). Input from outside is treated similarly. However, just replacing input by unconditionally enabled $skip$-actions would be unsound, because it renders potential $tick$-steps impossible by ignoring the situation when the chaotic environment does *not* send any message. The core of the problem is that with the timed semantics, a chaotic environment not just sends streams of messages, but "chaotically timed" message streams, i.e., with $tick$'s interspersed at arbitrary points.

We embed the chaotic nature of the environment by adding to each process P a new timer variable t_P, used to guard the input from outside.[2] These timers behave in the same manner as the old "chaotic" timers, except that we do not allow the new t_P timers to become deactivated (cf. rules T-INPUT$_2$ and T-NOINPUT). Since for both input and output, the communication statement using an external signal is replaced by a $skip$, the transformation yields a *closed* system.

The relationship between the intermediate and the transformed program will again be based on *simulation* (cf. Definition 6) but with different choices for the order relations on configurations and on labels. Based on the dataflow analysis, the transformation considers certain variable instances as potentially chaotic and unreliable. Hence to compare configurations of $S^{\mathbb{T}}$ and $S^\sharp$, we have to take η^α into account. So relative to a given analysis η^α, we define the relationship between valuations as follows: $\eta^\alpha \models \eta^{\mathbb{T}} \leq \eta^\sharp$,

[2] Note that the action $g_{t_P} \rhd reset\ t_P; set\ t_P := 0$ in rule T-INPUT$_2$ corresponds to the do-nothing step $g_{t_P} \rhd skip$.

$$\frac{l \longrightarrow_{g \,\triangleright\, x:=e} \hat{l} \in Edg^{\mathbb{T}} \qquad [\![e]\!]_{\eta_l^\alpha} \neq \top \qquad g^\sharp = [\![g]\!]_{\eta_l^\alpha}}{l \longrightarrow_{g^\sharp \,\triangleright\, x:=e} \hat{l} \in Edg^\sharp}\ \textrm{T-}\textsc{Assign}_1$$

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

$$\frac{l \longrightarrow_{?s(x)} \hat{l} \in Edg^{\mathbb{T}} \qquad s \notin Sig_{ext}}{l \longrightarrow_{?s(x)} \hat{l} \in Edg^\sharp}\ \textrm{T-}\textsc{Input}_1$$

$$\frac{l \longrightarrow_{?s(x)} \hat{l} \in Edg^{\mathbb{T}} \qquad s \in Sig_{ext}}{l \longrightarrow_{g_{t_P} \,\triangleright\, reset\ t_P} \longrightarrow_{set\ t_P:=0} \hat{l} \in Edg^\sharp}\ \textrm{T-}\textsc{Input}_2$$

$$\frac{}{l \longrightarrow_{g_{t_P} \,\triangleright\, reset\ t_P} \longrightarrow_{set\ t_P:=1} l \in Edg^\sharp}\ \textrm{T-}\textsc{NoInput}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, P'!(s,e)} \hat{l} \in Edg^{\mathbb{T}} \qquad s \notin Sig_{ext} \qquad g^\sharp = [\![g]\!]_{\eta_l^\alpha}}{l \longrightarrow_{g^\sharp \,\triangleright\, P'!(s,e)} \hat{l} \in Edg^\sharp}\ \textrm{T-}\textsc{Output}_1$$

$$\frac{l \longrightarrow_{g \,\triangleright\, P'!(s,e)} \hat{l} \in Edg^{\mathbb{T}} \qquad s \in Sig_{ext} \qquad g^\sharp = [\![g]\!]_{\eta_l^\alpha}}{l \longrightarrow_{g^\sharp \,\triangleright\, skip} \hat{l} \in Edg^\sharp}\ \textrm{T-}\textsc{Output}_2$$

$$\frac{l \longrightarrow_{g \,\triangleright\, set\ t:=e} \hat{l} \in Edg^{\mathbb{T}} \qquad g^\sharp = [\![g]\!]_{\eta_l^\alpha} \qquad [\![e]\!]_{\eta_l^\alpha} \neq \top}{l \longrightarrow_{g^\sharp \,\triangleright\, set\ t:=e} \hat{l} \in Edg^\sharp}\ \textrm{T-}\textsc{Set}_1$$

$$\frac{l \longrightarrow_{g \,\triangleright\, set\ t:=e} \hat{l} \in Edg^{\mathbb{T}} \qquad g^\sharp = [\![g]\!]_{\eta_l^\alpha} \qquad [\![e]\!]_{\eta_l^\alpha} = \top}{l \longrightarrow_{g^\sharp \,\triangleright\, set\ t:=0} \hat{l} \in Edg^\sharp}\ \textrm{T-}\textsc{Set}_2$$

$$\frac{l \longrightarrow_{g \,\triangleright\, reset\ t} \hat{l} \in Edg^{\mathbb{T}} \qquad g^\sharp = [\![g]\!]_{\eta_l^\alpha}}{l \longrightarrow_{g^\sharp \,\triangleright\, reset\ t} \hat{l} \in Edg^\sharp}\ \textrm{T-}\textsc{Reset}$$

$$\frac{l \longrightarrow_{g_t \,\triangleright\, reset\ t} \hat{l} \in Edg^{\mathbb{T}} \qquad g_t^\sharp = [\![g_t]\!]_{\eta_l^\alpha}}{l \longrightarrow_{g_t^\sharp \,\triangleright\, reset\ t} \hat{l} \in Edg^\sharp}\ \textrm{T-}\textsc{Timeout}$$

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

Table 5. Transformed system

iff for all variables $x \in Var$ one of the two conditions hold: $[\![x]\!]_{\eta^\top} = [\![x]\!]_{\eta^\sharp}$ or $[\![x]\!]_{\eta_l^\alpha} = \top$. Note that nothing is required for the new timer variables t_P.

The set of observable input signals of a process P is defined as

$$Sig^P_{obs} = \{s \in Sig \backslash Sig_{ext} \mid \neg \exists l \longrightarrow_{g \,\triangleright\, P!(s,e)} \hat{l}.\ [\![e]\!]_{\eta_l^\alpha} = \top\}.$$

The observable effect of input and output labels is given by the following equations:

$$\ulcorner P?(s,v)\urcorner = \begin{cases} \tau & \text{if } s \in Sig_{ext} \\ P?(s,v) & \text{if } s \in Sig^P_{obs} \\ P?(s) & \text{else} \end{cases} \qquad \ulcorner P!(s,v)\urcorner = \begin{cases} \tau & \text{if } s \in Sig_{ext} \\ P!(s,v) & \text{if } s \in Sig^P_{obs} \\ P!(s) & \text{else} \end{cases}$$

For $tick$- and τ-labels, $\ulcorner\cdot\urcorner$ acts as identity. With this definition, we choose as order relation on labels $\lambda^\top \leq \lambda^\sharp$ if $\ulcorner\lambda^\top\urcorner = \ulcorner\lambda^\sharp\urcorner$. In accordance with this definition, we set $\leq$ on the input queues of $P^\top$ and $P^\sharp$ inductively as follows: for empty queues $\epsilon \leq \epsilon$. In the induction case $(s, v^\top) :: q^\top \leq q^\sharp$, if $s \in Sig_{ext}$ and $q^\top \leq q^\sharp$. Otherwise, $(s, v^\top) :: q^\top \leq (s, v^\sharp) :: q^\sharp$, if $q^\top \leq q^\sharp$ and furthermore $\ulcorner(s, v^\top)\urcorner = \ulcorner(s, v^\sharp)\urcorner$, where $\ulcorner\cdot\urcorner$ for queue messages is defined in analogy to the definition for labels. This means, when comparing the queues, the *external* messages are ignored for $P^\sharp$, while for the internal messages, the signals must coincide and the value component is compared on the result of the analysis on the potential sending locations. The $\leq$-definitions are extended in the obvious manner to expressions and configurations.

In order to have the transformed system exhibit only more behaviour than the intermediate one, it must be guaranteed that whenever a guarded edge can be taken in $S^\top$, the corresponding guard for $S^\sharp$ likewise evaluates to $true$, where we have to take into account that in the intermediate level, guards with value $\top$ enable the action, as well. This property is an immediate consequence of the construction of the guards $g^\sharp$ in $S^\sharp$.

Lemma 9. *Assume two systems $S^\top$ and $S^\sharp$ and $\eta^\alpha \models (l, \eta^\top) \leq (l, \eta^\sharp)$.*

1. *Let g be a guard of an edge in $S^\top$ originating at location l and $g^\sharp$ its analogue in $S^\sharp$. If $[\![g]\!]_{\eta^\top} \in \{true, \top\}$, then $[\![g^\sharp]\!]_{\eta^\sharp} = true$.*
2. *If $[\![t]\!]_{\eta^\top} \in \{on(0), on(\top)\}$, then $[\![t]\!]_{\eta^\sharp} = on(0)$.*

Lemma 10. *Let $\gamma^\top = (l, \eta^\top, q^\top)$ be a configuration of $S^\top$. Then there exists an input-edge starting from l, or an edge guarded by g and where $[\![g]\!]_{\eta^\top} = true$ or $[\![g]\!]_{\eta^\top} = \top$.*

Lemma 11. *Let $\gamma^\top$ and $\gamma^\sharp$ be two configurations of $S^\top$ and $S^\sharp$, such that $\eta^\alpha \models \gamma^\top \leq \gamma^\sharp$. If $blocked(\gamma^\top)$, then $\gamma^\sharp = \gamma_0^\sharp \to_\tau \gamma_1^\sharp \to_\tau \ldots \to_\tau \gamma_n^\sharp = \hat{\gamma}^\sharp$ for some configurations $\gamma_i^\sharp$ and some $n \geq 0$ such that $\gamma^\top \leq \gamma_i^\sharp$ for all i, and $blocked(\hat{\gamma}^\sharp)$.*

Lemma 12. *Let $S^\top$ and $S^\sharp$ as well as $\leq$ be defined as above. Then $S^\top \preceq S^\sharp$.*

Proof sketch. With the help of Lemma 9, it is straightforward to check on the rules of Tables 1 and 4 together with the transformation rules, that for single processes $P^\top \preceq P^\sharp$. For systems, prove the implication that $P_1^\top \preceq P_1^\sharp$ and $P_2^\top \preceq P_2^\sharp$ implies $P_1^\top \parallel P_2^\top \preceq P_1^\sharp \parallel P_2^\sharp$, proceeding similarly by case analysis on the rules of Table 2. There, for the case of TICK, use Lemma 11. $\square$

Having established simulation between the two levels, we can proceed to the relationship we are really interested in, namely: the transformed systems must be a safe abstraction as far as the logic is concerned. Being in simulation relation guarantees preservation of LTL-properties as long as variables influenced by chaos are not mentioned. Therefore, we define as set of observable variables $Var_{obs} = \{x \mid \neg\exists l \in Loc.\ [\![x]\!]_{\eta_l^\alpha} = \top\}$. Note that the additional timer variables t_P are unobservable.

Lemma 13. *Let the relations $\leq$ and Var_{obs} be defined as above. Then for all formulas φ from next-free LTL, $S^\top \preceq S^\sharp$ and $S^\sharp \models \varphi$ implies $S^\top \models \varphi$.*

This brings us to the paper's final result: as immediate consequence of the above development, we obtain the desired property preservation:

Corollary 14. *Let S, $S^\sharp$, and Var_{obs} be defined as before, and φ a next-free LTL-formula mentioning only variables from Var_{obs}. Then $S^\sharp$ is closed and $S^\sharp \models \varphi$ implies $S \models \varphi$.*

5 Conclusion

In this paper, we apply dataflow analysis to transform an open system into a closed, safe abstraction, well-suited for model checking. The method of embedding chaos has been successfully applied in the context of the Vires project (*V*erifying *I*ndustrial *Re*active *S*ystems) [38]. To cope with the complexity of the project's verification case study, an industrial wireless ATM medium-access layer protocol (Mascara) [14, 39], we followed a compositional approach, which immediately incurred the problem of closing the modules [34, 35].

Related work Closing open (sub-)systems is common for software *testing*. In this field, a work close to ours in spirit and techniques is the one of [12]. It describes a dataflow algorithm to close program fragments given in the C-language with the most general environment and at the same time eliminating the external interface. The algorithm is incorporated into the *VeriSoft* tool. Similar to the work presented here, they assume an asynchronous communicating model, but do not consider *timed* systems and their abstraction. Similarly, [18] consider partial (i.e., open) systems which are transformed into closed ones. To enhance the precision of the abstraction, their approach allows to close the system by an external environment more specific than the most general, chaotic one, where the closing environment can be built to conform to given assumptions, which they call filtering [16]. As in our work, they use LTL as temporal logic and Spin as model checker, but the environment is modelled separately and is not embedded into the system.

A more fundamental approach to model checking open systems, also called reactive modules [3], is known as *module* checking [26][25]. 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 [5] is a model checker for reactive modules, which uses alternating-time temporal logic [4] as specification language.

Slicing, a well-known program analysis technique, resembles the analysis described in this paper, in that it is a data-flow analysis computing — in forward or backward direction — parts of the program that may depend on the certain points of interest (cf. for a survey [36]). The analysis of Section 3 computes in a forward manner the cone of influence of all points of the system influenced from the outside. The usefulness of slicing for model checking is explored in [29], where slicing is used to speed up model checking and simulation for programs in Promela, Spin's input language. However, the program transformation in [29] is not intended to preserve program properties in general. Likewise in the context of LTL model checking, [17] use slicing to cut away irrelevant program fragments but the transformation yields a safe, property-preserving abstraction and potentially a smaller state space.

Future work While chaos is useful as the most abstract approximation of the environment, one often can verify properties of a component only under assumptions or restrictions on the environment behaviour. For future work we plan to generalize the framework to embed also environments given by timed LTL-formulas. For timers, a more concrete behaviour than just using random expiration periods could be automatically extracted from the sub-components by data-flow techniques, leading to more refined timer abstraction.

References

1. ACM. *First Annual Symposium on Principles of Programming Languages (POPL) (Boston, MA)*, January 1973.
2. ACM. *Twelfth Annual Symposium on Principles of Programming Languages (POPL)*, January 1985.
3. R. Alur and T. A. Henzinger. Reactive modules. In *Proceedings of LICS '96*, pages 207–218. IEEE, Computer Society Press, July 1996.
4. R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proceedings of the IEEE Symposium on Foundations of Computer Science, Florida*, Oct. 1997.
5. 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.
6. D. Bošnački and D. Dams. Integrating real time into Spin: A prototype implementation. In Budkowski et al. [9].
7. D. Bošnački, D. Dams, L. Holenderski, and N. Sidorova. Verifying SDL in Spin. In Graf and Schwartzbach [20].
8. E. Brinksma, editor. *International Workshop on Protocol Specification, Testing and Verification IX*. North-Holland, 1989. IFIP TC-6 International Workshop.
9. 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.
10. 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.

11. E. M. Clarke and R. P. Kurshan, editors. *Computer Aided Verification 1990*, volume 531 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.
12. 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.
13. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems: Abstraction preserving $\forall$CTL*,$\exists$CTL*, and CTL*. In Olderog [31].
14. I. Dravapoulos, N. Pronios, A. Andristou, I. Piveropoulos, N. Passas, D. Skyrianoglou, G. Awater, J. Kruys, N. Nikaein, A. Enout, S. Decrauzat, T. Kaltenschnee, T. Schumann, J. Meierhofer, S. Thömel, and J. Mikkonen. *The Magic WAND, Deliverable 3D5, Wireless ATM MAC, Final Report*, Aug. 1998.
15. Discrete-time Spin. `http://win.tue.nl/~dragan/DTSpin.html`, 2000.
16. M. Dwyer and D. Schmidt. Limiting state explosion with filter-based refinement. In *Proceedings of the 1st Internaltional Workshop in Verification, Abstract Interpretation, and Model Checking*, Oct. 1997.
17. M. B. Dwyer and J. Hatcliff. Slicing software for model construction. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, Jan. 1999.
18. M. B. Dwyer and C. S. Pasareanu. Filter-based model checking of partial systems. In *Proceedings of the 6th ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT '98)*, pages 189–202, 1998.
19. P. Godefroid. Using partial orders to improve automatic verification methods. In Clarke and Kurshan [11], pages 176–449. an extended Version appeared in ACM/AMS DIMACS Series, volume 3, pages 321–340, 1991.
20. S. Graf and M. Schwartzbach, editors. *Proceedings of the Sixth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, volume 1785 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
21. M. S. Hecht. *Flow Analysis of Programs*. North-Holland, 1977.
22. G. Holzmann and J. Patti. Validating SDL specifications: an experiment. In Brinksma [8], pages 317–326. IFIP TC-6 International Workshop.
23. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
24. G. Kildall. A unified approach to global program optimization. In *Proceedings of POPL '73* [1], pages 194–206.
25. 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.
26. 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.
27. 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)* [2], pages 97–107.
28. D. Long. *Model Checking, Abstraction and Compositional Verification*. PhD thesis, Carnegie Mellon University, 1993.
29. L. I. Millet and T. Teitelbaum. Slicing promela and its application to model checking, simulation, and protocol understanding. In *Electronic Proceedings of the Fourth International SPIN Workshop, Paris, France*, Nov. 1998.
30. F. Nielson, H.-R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
31. E.-R. Olderog, editor. *Working Conference on Programming Concepts, Methods and Calculi, San Miniato, Italy*. IFIP, North-Holland, June 1994.

32. A. Pnueli. The temporal logic of programs. In *Proceeding of the 18th Annual Symposium on Foundations of Computer Science*, pages 45–57, 1977.
33. Specification and Description Language SDL, blue book. CCITT Recommendation Z.100, 1992.
34. N. Sidorova and M. Steffen. Verification of a wireless ATM medium-access protocol. In *Proceedings of the 7th Asia-Pacific Software Engineering Conference (APSEC 2000), 5.–8. December 2000, Singapore*, pages 84–91. IEEE Computer Society, 2000. A preliminary and longer version appeared as Universität Kiel technical report TR-ST-00-3.
35. N. Sidorova and M. Steffen. Verifying large SDL-specifications using model checking. Feb. 2001. To appear in the LNCS-proceedings of the 10th SDL-Forum 2001 "Meeting UML".
36. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.
37. 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.
38. Verifying industial reactive systems (VIRES), Esprit long-term research project LTR-23498. `http://radon.ics.ele.tue.nl/~vires/`, 1998-2000.
39. A wireless ATM network demonstrator (WAND), ACTS project AC085. `http://www.tik.ee.ethz.ch/~wand/`, 1998.