Closing open SDL-systems for model checking with DTSpin

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

[1] Department of Software Engineering, CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Natalia.Ioustinova@cwi.nl

[2] Department of Mathematics and Computer Science
Eindhoven University of Technology
Den Dolech 2, P.O. Box 513,
5612 MB Eindhoven, The Netherlands
n.sidorova@tue.nl

[3] Institute of Computer Science and Applied Mathematics
Christian-Albrechts-Universität
Preußerstraße 1–9,
24105 Kiel, Germany
ms@informatik.uni-kiel.de

Abstract. Model checkers like Spin can handle closed reactive systems, only. Thus to handle open systems, in particular when using assume-guarantee reasoning, we need to be able to close (sub-)systems, which is commonly done by adding an environment process. For models with asynchronous message-passing communication, however, modelling the environment as separate process will lead to a combinatorial explosion caused by all combinations of messages in the input queues.
In this paper we describe the implementation of a tool which automatically closes DTPromela translations of SDL-specifications by embedding the timed chaotic environment into the system. To corroborate the usefulness of our approach, we compare the state space of models closed by embedding chaos with the state space of the same models closed with chaos as external environment process on some simple models and on a case study from a wireless ATM medium-access protocol.
Keywords: model checking, SDL, DTSpin, open communication systems, abstractions.

1 Introduction

Model checking is becoming an increasingly important part of the software design process [10]. Modern commercial SDL design tools like ObjectGeode [30] and the TAU SDL suite [1] allow validation of SDL specifications through simulation and testing. Since errors in the telecommunication systems are expensive, there is a need for additional ways of verification and debugging, and model checking of SDL specifications is an area of active research, cf. e.g. [19, 6, 4, 21, 22, 18, 35].

* Author supported by the "Systems Validation Centre (SVC)" project

Despite all algorithmic advances in model checking techniques and progress in raw computing power, however, the state explosion problem limits the applicability of model-checking [8, 31, 9] and thus *decomposition* and *abstraction* are indispensable when confronted with checking large designs. Following a decompositional approach and after singling out a subcomponent to check in isolation, the next step often is to *close* the subcomponent with an environment, since most model checkers cannot handle open systems.

Closing is generally done by adding an overapproximation of the real environment in the form of an external process. To allow the transfer of positive verification results from the constructed closed model to the real system, the environment process must be a safe abstraction [12, 13] of the real environment, i.e., it must exhibit at least all the behaviour of the real environment. In the simplest case this means the closing environment behaves *chaotically.*

In an asynchronous communication model, just adding an external chaos process will not work, since injecting arbitrary message streams to the unbounded input queues will immediately lead to an infinite state space, unless some restrictions on the environment behaviour or on the maximal queue length are imposed in the closing process. Even so, external chaos results in a combinatorial explosion caused by all combinations of messages in the input queues.

In [32], we describe a simple approach which avoids the state-space penalty in the queues by "embedding" the external chaos into the component under consideration. We use *data abstraction,* condensing data from outside into a single abstract value to deal with the infinity of environmental data. 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. Furthermore, since we are dealing with the discrete-time semantics [22, 4] 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. Using the result of the analysis, the transformation yields a *closed* system which is a safe abstraction of the original one in terms of traces.

Based on these earlier theoretical results, the main contribution of this paper is the description of a tool implementing the embedded closing ideas and the presentation of experimental results that corroborate the usefulness of the approach. The implementation is targeted towards the verification with DTSpin, a discrete time extension of the well-known Spin model checker, therefore we chose to close DTPromela translations of SDL specifications. The experiments performed with DTSpin confirmed that the proposed method leads to a significant reduction of the state space and the verification time.

The rest of the paper is organized as follows. In the following Section 2, we sketch the formal background of the method. Afterwards, in Sections 3 and 4, we present the toolset we use, its extension, and the experimental results of a few smaller examples as well as the results on a larger case study. We conclude in Section 5 with discussing related work.

2 Embedding chaos

In this section, we recapitulate the ideas underlying the program transformation to yield a closed system. A more detailed account of the underlying theory can be found in [32]. We start with fixing syntax and semantics and proceed with program transformation and data-flow analysis required for the transformation.

2.1 Semantics

Our operational model is based on asynchronously communicating state machines (processes) with top-level concurrency. Since we take SDL as a source language and DTPromela as a target one, the operational model gives the semantics of a subset of SDL that does not allow procedure calls and dynamic process creation, and also suits as semantics for a subset of DTPromela that is a target of translation from IF to DTPromela.

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*; 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 $?s(x)$, $g \rhd P!s(e)$, 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 $l \longrightarrow_{\alpha} \hat{l}$.

Time aspects of a system behaviour are specified by actions dealing with *timers*. 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 guards. The equivalence of timeouts-by-guards and timeouts-as-messages in the presence of SDL's asynchronous communication model is argued for in [4]. The time semantics chosen here is not the only one conceivable (see e.g. [7] for a broader discussion of the use of timers in SDL). The semantics we use is the one described in [22, 4], and is also implemented in DTSpin [3, 14].

Each process has a finite set of timer variables (with typical elements $t, t'_1, \dots$) 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 $on(v)$ (rule SET), i.e., it is activated to run for the designated period, or deactivated (rule RESET)), i.e., it has a value *off*. 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 (rule TIMEOUT). A possible discard of a timeout signal is imitated by analogous action (rule TDISCARD).

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 $(\sigma_{init}, \epsilon) = (\sigma_0, q_0) \rightarrow_\lambda (\sigma_1, q_1) \rightarrow_\lambda \dots$ 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 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.

An input of a signal is enabled if the signal at the head of the queue matches signal expected by the process. Inputting results in removing the signal from the head of the queue and updating the local valuation according to parameters of the signal. In rule INPUT $\eta \in Val$, 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. The rule DISCARD captures a specific feature of SDL92: if the signal from the head of the queue does not match any input defined as possible for the current (input) location, the signal is removed from the queue without changing the location and the valuation. Unlike input, output is guarded, so sending a message involves evaluating the guard and the expression according to the current valuation (rule OUTPUT). In OUTPUT P' stands for the process identity of the destination and P is the identity of the sender. 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 and P' is the identity of a sender. We assume for the non-timer guards, that at least one of them evaluates to true for each configuration.

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

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

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

$$\frac{v \in D}{(l, \eta, q) \rightarrow_{P?P'(s,v)} (l, \eta, q :: (s, v))}\ \textsc{Receive}$$

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

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

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

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

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

Table 1. Step semantics for process P

global transitions synchronize via their common labels. The global step relation $\rightarrow_\lambda \subseteq \mathit{\Gamma} \times \mathit{Lab} \times \mathit{\Gamma}$ is given by the rules of Table 2.

Asynchronous communication between the two processes uses signal s to exchange a common value v, as given by rule Comm. The communication takes place led for matching identities of sender and receiver process. For τ-steps and non-matching communication steps are concerned, each process can proceed on its own by the interleaving rules; each of these rules has 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 $\textsc{Tick}_P$, this is expressed by the predicate *blocked* on configurations: $\mathit{blocked}(\sigma)$ holds if no move is possible by the system except either a clock-tick or a reception of a message from the outside. Note in passing that due to the discarding feature, $\mathit{blocked}(\sigma, q)$ 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., $\mathit{on}(n+1) - 1 = \mathit{on}(n)$, non-active timers are not affected. Note that the operation is undefined for $\mathit{on}(0)$, since a configuration can perform a tick only if not timer equals $\mathit{on}(0)$.

$$\frac{(\sigma_1, q_1) \rightarrow_{P_2!P_1(s,v)} (\hat{\sigma}_1, \hat{q}_1) \qquad (\sigma_2, q_2) \rightarrow_{P_2?P_1(s,v)} (\hat{\sigma}_2, \hat{q}_2)}{(\sigma_1, q_1) \times (\sigma_2, q_2) \rightarrow_{\tau} (\hat{\sigma}_1, \hat{q}_1) \times (\hat{\sigma}_2, \hat{q}_2)} \text{ COMM}$$

$$\frac{(\sigma_1, q_1) \rightarrow_{P_1?P_2'(s,v)} (\hat{\sigma}_1, \hat{q}_1) \qquad P_2' \neq P_2}{(\sigma_1, q_1) \times (\sigma_2, q_2) \rightarrow_{P_1?P_2'(s,v)} (\hat{\sigma}_1, \hat{q}_1) \times (\sigma_2, q_2)} \text{ INTERLEAVE}_1$$

$$\frac{(\sigma_1, q_1) \rightarrow_{P_2'!P_1(s,v)} (\hat{\sigma}_1, \hat{q}_1) \qquad P_2' \neq P_2}{(\sigma_1, q_1) \times (\sigma_2, q_2) \rightarrow_{P_2'!P_1(s,v)} (\hat{\sigma}_1, \hat{q}_1) \times (\sigma_2, q_2)} \text{ INTERLEAVE}_2$$

$$\frac{(\sigma_1, q_1) \rightarrow_{\tau} (\hat{\sigma}_1, \hat{q}_1)}{(\sigma_1, q_1) \times (\sigma_2, q_2) \rightarrow_{\tau} (\hat{\sigma}_1, \hat{q}_1) \times (\sigma_2, q_2)} \text{ INTERLEAVE}_{\tau}$$

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

Table 2. Parallel composition of P_1 and P_2

2.2 Abstracting data

Next we present a straightforward dataflow analysis marking variable and timer instances that may be influenced by the environment.

The analysis uses 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 $Val^\alpha = Var \rightarrow \{\top, \bot\}$. 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 finite 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 $\bigvee_{i=1}^n \eta_i^\alpha$.

Each node n of the flow graph has associated an abstract transfer function $f_n : Val^\alpha \rightarrow Val^\alpha$. The functions are given in Table 3, where α_n denotes the action associated with the node n. The equations describe the change of the abstract valuations depending on the sort of the 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 and where Sig_{ext} are the signals potentially sent by the environment. 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: $\eta_{init}^\alpha(n) = \eta_\bot$. We are

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

Table 3. Transfer functions/abstract effect for process P

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

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

For each node n of the flow graph, the data-flow problem is specified by two inequations or constraints. The first one relates the abstract valuation η^α_{pre} before entering the node with the valuation η^α_{post} afterwards via the abstract effects of Table 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, 20, 29]), where the worklist steers the iterative loop until the least fixpoint is reached (cf. Fig. 1).

---

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

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

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

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

Fig. 1. Worklist algorithm

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 the unfinished one. After termination the algorithm yields two mappings $\eta^{\alpha}_{pre}, \eta^{\alpha}_{post} : \mathit{Node} \rightarrow \mathit{Val}^{\alpha}$. 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}.

2.3 Program transformation

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

The transformation is given as a set of transformation rules (see Table 4) for each process P. The transformation is straightforward: guards potentially influenced by the environment are taken non-deterministically, i.e., a guard g at a location l replaced by $true$, if $[\![g]\!]_{\eta^{\alpha}_{l}} = \top$. Assignments of expressions are either left untouched or replaced by *skip*, depending on the result of the analysis concerning the left-hand value of the assignment (rules T-ASSIGN$_1$ and T-ASSIGN$_2$). 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 behaviour of the timer expiration is captured by arbitrary postponing a timeout by setting back the value of the timer to $on(1)$ according to T-NOTIMEOUT.

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.[1] These timers behave in the same manner as the "chaotic" timers above, 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. Outputs to the environment are just removed (rule T-OUTPUT$_2$).

3 Extending the Vires toolset

The Vires toolset was introduced for verification of industrial-size communication protocols. Its architecture is targeted towards the verification of SDL specifications and it provides an automatic translation of SDL-code into the input language of a discrete-time extension of the well-known Spin model-checker.

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

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

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

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

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

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

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

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

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

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

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

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

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

Table 4. Transformed system

Design, analysis, verification, and validation of SDL specifications is supported by OBJECTGEODE, one of the most advanced integrated SDL-environments. OBJECTGEODE also provides code generation and testing of real-time and distributed applications.

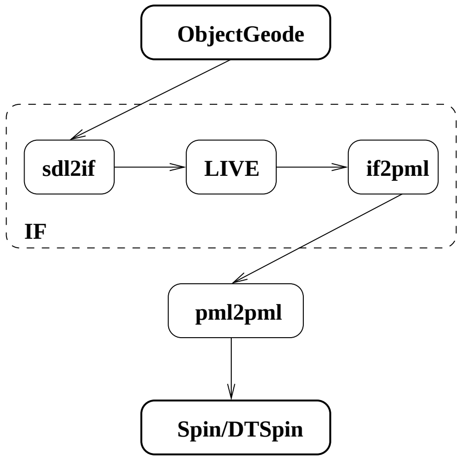

Fig. 2. Toolset components

Spin [23] is a state-of-the-art, enumerative model-checker with an expressive input-language Promela. In an extensive list of industrial applications, Spin and Promela have proven to be useful for the verification of industrial systems. Spin can be used not only as a simulator for rapid prototyping that supports random, guided and interactive simulation, but also as a powerful state space analyzer for proving user-specified correctness properties of the system. As standard Spin does not deal with timing aspects of protocols, DTSpin, a discrete time extension of Spin has been developed [3, 14], that can be used for verification of properties depending on timing parameters. The extension is compatible with the standard untimed version of the Spin validator, except for the timeout statement, which has different semantics and its usage is no longer allowed (nor necessary) in discrete-time models.

IF [5] bridges the gap between OBJECTGEODE and Spin/DTSpin. It contains a translator, SDL2IF of SDL specifications into the intermediate representation IF. A static analyzer Live [27] performs optimization of IF-representation to reduce the state space of the model. IF-specifications can be translated to DTPromela models with the help of IF2PML-translator [4] and verified by DTSpin.

The PML2PML-translator takes care of the automatic closing of a subcomponent and implements the theory presented before. The tool post-processes the output from the translation from the SDL-specification to Promela, where the implementation covers the subset of SDL described abstractly in Section 2.

The translator works fully automatic and does not require any user interaction, except that the user is required to indicate the list of external signals. The extension is implemented in Java and requires JDK-1.2 or later. The package can be downloaded from `http://www.cwi.nl/~ustin/EH.html`.

4 Experimental results

Before we present the results on a larger example — the control-part of a medium-access protocol — we show the effect of the transformation on the state space using a few artificial, small examples.

4.1 Simple motivating examples

In this subsection we take some simple open systems modelled in DTPromela, close them with chaos as separate process and illustrate how the state space grows with the buffer length and with the number of signals involved into the communication with the environment.

First, we construct a DTPromela model of a process that receives signals a, b, and c from the outside, and reacts by sending back d, e, and f, respectively.

```
proctype proc(){
    start: goto q;
    q: atomic{ if
                :: envch?a ->  proch!d; goto q;
                :: envch?b ->  proch!e; goto q;
                :: envch?c ->  proch!f; goto q; fi;
                 }
    }
```

A closing environment will send the messages a, b, and c to the process, and conversely receive d, e, and f in an arbitrary manner. As explained in Section 2, the environment must behave chaotically also wrt. the timing behaviour. Therefore, in order to avoid zero-time cycles, the sending actions are guarded by a timeout and an extra clause is added when no more signals are to be sent in the current time slice. A specification of such an environment process is given below:

```
     s: atomic{ if
          :: expire(t) -> set(t, 1); goto s; /* stop sending
                          signals until the next time slice */
          :: expire(t) -> envch!a; set(t, 0); goto s;
          ...........
          :: proch?f -> goto s;
          fi }
    }
```

The queues in the verification model, however, have to be bounded. There are two options in Spin for handling queues. The first one is to block a process attempting to send a message to a full queue until there is a free cell in the queue. With this option, our "naive" closing leads to a deadlock caused by an attempt of a process to send a message to the full queue of the environment while the environment is trying to send a message to the full process queue. Another option is to lose new messages in case the queue is full. In this case large number of messages gets lost (see Table 5). Many properties cannot be verified using this option. Moreover, there is a large class of systems where messages should not get lost, for this would lead to non-realistic behaviour of the system. Still, even when this option is applicable, time and memory consumption grow tremendously fast with the buffer size, as shown in Table 5.

option	buffer	states	transitions	lost messages	memory (MB)	time
block	3	deadlock				
loose	3	3783	13201	5086	2.644	00.24 s
loose	4	37956	128079	47173	3.976	01.97 s
loose	5	357015	1.18841e+06	428165	18.936	20.49 s
loose	6	3.27769e+06	1.08437e+07	3.86926e+06	170.487	4 min 04.74 s

Table 5. Different buffer sizes, unlimited number of signals per time slice

We can avoid the deadlock in the system above if we limit a number of messages sent by the environment per a time slice. For this purpose we introduce an integer variable n set to the queue size and modify the options of the *if* statement in such a way that sendings are enabled only if n is positive; n is counted down with every message sent and n is revived every time before a new time slice starts.

```
:: (n>0 && expire(t)) -> envch!a; n = n-1; set(t, 0); goto ea;
........................
:: expire(t) -> set(t, 1); n= BUFFSIZE; goto ea;
```

Verification results for the system closed in such a way are shown in Table 6. And again, though more slowly than in the previous example, the number of states, transitions, memory usage, and time required for the verification grow with the queue length very fast.

Next we fix the length of the queue at 4 and vary the number of different messages sent from the process to the environment and from the environment to the process. Table 7 shows the experimental results. Note that the growth of the state space of the system is now caused by the combinatorial explosion in the queues. (The maximal number of messages that can be sent per a time slice is still equal to the length of the queue.)

In the experiments for the same process with the environment embedded and not external, the number of states is constant for all the cases considered

option	buffer	states	transitions	memory (MB)	time
block	3	328	770	2.542	00.06 s
block	4	1280	3243	2.542	00.10 s
block	5	4743	12601	2.747	00.24 s
block	6	16954	46502	3.259	00.78 s

Table 6. Different buffer sizes (4 signals per time slice)

n-messages	states	transitions	memory (MB)	time
4	3568	9041	2.644	00.22 s
5	8108	20519	2.849	00.42 s
6	16052	40569	3.156	00.75 s
7	28792	72683	3.771	01.36 s
8	47960	120953	4.590	02.45 s
9	75428	190071	5.819	03.86 s

Table 7. Different numbers of message types

and equal to 4. As one might have expected, closing system by a separate environment process behaving chaotically, leads to a state space explosion even for very simple small systems. Tailoring the environment process such that only "relevant" messages can be sent makes the environment process large and complicated, which can also cause the growth of the state space or lead to errors caused by mistakes in the environment design.

4.2 Case study: a wireless ATM medium-access protocol

To validate our approach, we applied the PML2PML-translator in a series of experiments to the industrial protocol Mascara [36].

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

Composed of various protocol layers and sub-entities, Mascara is a large protocol. With the current state-of-the-art in automatic verification it is not possible to model check it as a whole — the compositional approach and abstractions are necessary. Since the model of Mascara is not trivial, already the state space of the obtained submodels with only several processes is large.

This protocol was the main case study in the Vires project; the results of its verification can be found e.g. in [4, 19, 33]. Here, we are not interested in the verification of Mascara's properties but in the comparison of the state space of a model of the *Mascara control* entity (MCL) at the mobile terminal side when closed with the environment as a separate chaotic process and the state space of the same entity closed with embedded chaos.

The *Mascara control* entity is responsible for the protocol's control and signaling tasks. It offers its services to the ATM-layer above while using the services of the underlying segmentation and reassembly entity, the sliding-window entities, and in general the low-layer data-pump. It carries out the periodical monitoring of the current radio link quality, gathering the information about radio link qualities of its neighbouring access points to be able to handover to one of them quickly in the case of deterioration of the current association link quality, and switching from one access point to another in the handover procedure.

In [33] we were closing MCL by embedding the chaotic environment *manually.* Not surprisingly, verifying properties of MCL closed with chaos yielded false negatives first in many cases — the completely chaotic environment was too abstract. Therefore, the traces leading to these false negatives were analyzed, which resulted in a refined environment. The refinement was done by identifying signals that could not be sent chaotically lest the verification property was violated, constructing a specific environment process sending only these signals, and closing the obtained still open system by embedding the residual chaos. The conditions imposed on sending the detached signals are in fact the conditions imposed on the behaviour of the rest of the protocol, which formed later the verification properties for the other protocol entities. Thus, by constructing the environment process we only produce an abstraction of the real environment, keeping it as abstract as possible and leaving the whole model still open, which means that the environment prescribes the order of sendings and receivings for a part of signals, only. In this way, we could still benefit from embedding the chaos into the process.

Of course, closing the system manually is time-consuming and error-prone. With the implemented translator, it became possible to reproduce the same series of experiments quickly, without looking for typos and omissions introduced during the manual closing. Moreover, we performed the same experiments for MCL closed with the chaotic environment modelled as a process. In our experiments we used DTSpin version 0.1.1, an extension of Spin 3.3.10, using the partial-order reduction and compression options. All the experiments were run on a Silicon Graphics Origin 2000 server on a single R10000/250MHz CPU with 8GB of main memory. Our aim was to compare the state space and resource consumption for the two closing approaches. Therefore, we did not verify any LTL properties.

Table 8 gives the results for the model checking of MCL with chaos as external process on the left and embedded on the right. The first column gives the buffer size for process queues. The other columns give the number of states, transitions, memory and time consumption, respectively. As one can see, the

bs	states	transitions	mem.	time	states	transitions	mem.	time
2	9.73e+05	3.64e+06	40.842	15:57.34	300062	1.06e+06	9.071	1:12.64
3	5.24e+06	2.02e+07	398.933	22:28.36	396333	1.85e+06	11.939	1:36.87
4	2.69e+07	1.05e+08	944.440	1:59:39.76	467555	2.30e+06	14.499	2:12.91

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

state space as well as the time and the memory consumption are significantly larger for the model with the environment as a process, and they grow with the buffer size much faster than for the model with embedded chaos. The model with embedded environment has a relatively stable state-space size and other verification characteristics.

5 Conclusion

In this paper we described the implementation of a tool which allows to automatically close DTPromela translations of SDL-specifications by embedding the timed chaotic environment into the system. Our experiments performed on the Mascara case study show the efficiency of the chaos closing method.

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 [11]. It describes a dataflow algorithm for closing program fragments given in the C-language with the most general environment, eliminating the external interface at the same time. 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, [17] 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 [15]. A more fundamental approach to model checking open systems 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 [2] is a model checker for reactive modules, which uses alternating-time temporal logic as specification language.

In the context of the IF-toolset [5], live variable analysis has been proven useful [27] to counter the state explosion. Slicing, a well-known program analysis technique, which resembles the analysis described in this paper, is explored in [28] to speed up model checking and simulation in Spin. Likewise in the context of LTL model checking, [16] use slicing to cut away irrelevant program fragments but the transformation yields a safe, property-preserving abstraction and potentially a smaller state space.

For the future, we will extend the subset of SDL our translator can handle, including complex data types, procedures and process creation. Based on the results from [34], another direction for future work is to the extend the PML2PML implementation to handle environments more refined than just chaos with building an environment process communicating to the system synchronously.

References

1. Telelogic TAU SDL Suite. http://www.telelogic.com/products/sdl/, 2002.
2. 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.
3. 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.
4. 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.
5. M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm, and L. Mounier. IF: An intermediate representation and validation environment for timed asynchronous systems. In J. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of Symposium on Formal Methods (FM 99)*, volume 1708 of *Lecture Notes in Computer Science*. Springer-Verlag, Sept. 1999.
6. M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm, and L. Mounier. IF: A validation environment for timed asynchronous systems. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of CAV '00*, volume 1855 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
7. 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.
8. E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994. A preliminary version appeared in the Proceedings of POPL 92.
9. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronisation skeletons using branching time temporal logic specifications. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 244–263. Springer-Verlag, 1982.
10. E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, Dec. 1996. Available also as Carnegie Mellon University technical report CMU-CS-96-178.
11. 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.
12. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximaton of fixpoints. In *Fourth Annual Symposium on Principles of Programming Languages (POPL) (Los Angeles, Ca)*, pages 238–252. ACM, January 1977.
13. 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.

14. Discrete-time Spin. `http://win.tue.nl/~dragan/DTSpin.html`, 2000.
15. M. Dwyer and D. Schmidt. Limiting state explosion with filter-based refinement. In *Proceedings of the 1st International Workshop in Verification, Abstract Interpretation, and Model Checking*, Oct. 1997.
16. 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.
17. 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.
18. A. B. F. Regensburger. Formal verification of SDL systems at the Siemens mobile phone department. In B. Steffen, editor, *Proceedings of TACAS '98*, number 1384 in Lecture Notes in Computer Science, pages 439–455. Springer-Verlag, 1998.
19. J. Guoping and S. Graf. Verification experiments on the Mascara protocol. In M. B. Dwyer, editor, *Model Checking Software, Proceedings of the 8th International SPIN Workshop (SPIN 2001), Toronto, Canada*, Lecture Notes in Computer Science, pages 123–142. Springer-Verlag, 2001.
20. M. S. Hecht. *Flow Analysis of Programs.* North-Holland, 1977.
21. U. Hinkel. Verification of SDL specifications on the basis of stream semantics. In Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors, *Proceedings of the 1st Workshop of the SDL Forum Society on SDL and MSC (SAM'98)*, pages 241–250, 1998.
22. 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.
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*, pages 194–206. ACM, January 1973.
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. L. G. M. Bozga, J. Cl. Fernandez. State space reduction based on Live. In A. Cortesi and G. Filé, editors, *Proceedings of SAS '99*, volume 1694 of *Lecture Notes in Computer Science.* Springer-Verlag, 1999.
28. L. I. Millet and T. Teitelbaum. Slicing promela and its application to model checking, simulation, and protocol understanding. In E. Najm, A. Serhrouchni, and G. Holzmann, editors, *Electronic Proceedings of the Fourth International SPIN Workshop, Paris, France*, Nov. 1998.
29. F. Nielson, H.-R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.
30. ObjectGeode 4. `http://www.csverilog.com/products/geode.htm`, 2000.
31. J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Proceedings of the 5th International Symposium on Programming 1981*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1982.

32. 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.
33. 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.
34. N. Sidorova and M. Steffen. Synchronous closing of timed SDL systems for model checking. In A. Cortesi, editor, *Proceedings of the hird International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI) 2002*, volume 2294 of *Lecture Notes in Computer Science*, pages 79–93. Springer-Verlag, 2002.
35. H. Tuominen. Embedding a dialect of SDL in Promela. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking, Proceedings of 5th and 6th International SPIN Workshops, Trento/-Toulouse*, volume 1680 of *Lecture Notes in Computer Science*, pages 245–260. Springer-Verlag, 1999.
36. A wireless ATM network demonstrator (WAND), ACTS project AC085. `http://www.tik.ee.ethz.ch/~wand/`, 1998.