Abstraction and Flow Analysis for Model Checking Open Asynchronous Systems

Natalia Ioustinova*
Dept. of Software Engineering,
CWI (Centrum voor Wiskunde en Informatica)
Kruislaan 413, P.O. Box 94079,
1090 GB Amsterdam, The Netherlands
Natalia.Ioustinova@cwi.nl

Natalia Sidorova
Dept. of Mathematics and Computer Science
Eindhoven University of Technology
Den Dolech 2, P.O.Box 513
5612 MB Eindhoven, The Netherland
n.sidorova@tue.nl

Martin Steffen
Inst. für angewandte Mathematik und Informatik
Christian-Albrechts-Universität
Preußerstraße 1–9
24105 Kiel, Deutschland
ms@informatik.uni-kiel.de

Abstract

Formal methods, especially model checking, are an indispensable part of the software engineering process. With large software systems currently beyond the range of fully automatic verification, however, a combination of decomposition and abstraction techniques is needed. To model check components of a system, a standard approach is to close the component with an abstraction of its environment. To make it useful in practice, the closing of the component should be automatic, both for data and for control abstraction. Specifically for model checking asynchronous open systems, external input queues should be removed, as they are a potential source of a combinatorial state explosion.

In this paper, we close a component synchronously by embedding the external environment directly into the system to avoid the external queues, while for the data, *we use a two-valued abstraction, namely data influenced from the outside or not. This gives a more precise analysis than the one investigated in [8]. To further combat the state explosion problem, we combine this data abstraction with a static analysis to remove superfluous code fragments. The static analysis we use is reminiscent to the one presented in [8], but we use a combination of a* may *and a* must*-analysis instead of a* may*-analysis.*

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

*Supported by the CWI-project "Systems Validation Centre (SVC)".

1. Introduction

Model checking techniques have proven useful for the verification of communication systems. There exist efficient algorithms to verify a system against correctness requirements expressed by system invariants or by formulas of a temporal logic. The size of the model often grows exponentially with the number of parallel system components, which limits the applicability of model checking techniques, and various techniques like decomposition and abstraction have been developed to mitigate the state explosion problem.

Using decomposition, one breaks up the system into several subsystems or components, verified separately. These components rely on the communication with the rest of the system, i.e., they are open. Many model checkers, e.g. Spin [7], do not handle open systems and hence one needs to close open subsystems prior to verification.

Asynchronously communicating software systems such as communication protocols call for additional, specific optimization techniques to deal with the asynchronous nature of communication. In [13] we formalized a program transformation based on static analysis which takes the most abstract, i.e., chaotic environment, and "embeds" it into the component. Embedding the external chaos eliminates the need to explore the combinatorial state space of the external queues.

Part of the approach is the abstraction of environmental *data,* where, assuming a chaotic environment, a single abstract value is used. (The part of the abstraction dealing with timers does not interest us for the sake of this introduction.) Interested in a fully-automatic approach, [13] stressed efficiency over precision of abstraction, and we used a *static* data-flow analysis to mark all instances of variables *potentially* influenced from outside as chaotic, and to transform the program according to this reckoning.

In this paper, we improve on this abstraction in the following way. Instead of abstracting all potentially chaotic data in a static transformation, we use dynamically an abstract chaotic data value, which is represented in the implementation by the two-valued data-abstraction —data influenced by the environment or not— which gives a more precise approximation and hence less false negatives in the verification.

To further combat the state explosion problem, we combine this data abstraction with a static analysis *afterwards* to remove superfluous code fragments in the abstracted program. The static analysis we use is reminiscent to the one presented in [8], except that here we use a *must*-analysis instead of a *may*-analysis. The must analysis marks data definitely not influenced from outside, i.e., reliable data, and data definitely influenced; the rest forms then a "don't know" intermediate value for instances at those process locations where both chaotic and non-chaotic values can occur, depending on the system run leading to this instance. For optimization, we further transform the system in that we remove parts definitely influenced from outside and do not transform parts which are definitely not influenced.

Typical practical applications we are interested in are protocol specifications in SDL [11] and Promela [7]. More concretely, the developed methods for closing open asynchronous systems are used to automate the model checking of translations of SDL-specifications into DTPromela, the input language of the discrete-time Spin— model checker DTSpin.[1]

The remainder of the paper is organized as follows. In Section 2 we give a short overview of the semantics we work with. Section 3 defines the transformation rules and Section 4 contains the data-flow analysis for marking chaotic, non-chaotic, and not-known instances of variables to optimize the transformed system. In Section 5 we consider some examples to illustrate the approach and compare the transformation results with the ones obtained with the old approach from [13]. We conclude mentioning future and related work in Section 6.

[1]See [8] for details about the Vires toolset for the verification of DT-Promela translations of SDL-specifications.

2. Semantics

This section gives a brief overview of the structural operational semantics we work with. A more detailed account can be found in [8]. Our operational model is based on asynchronously communicating state machines or processes with top-level concurrency. Though the model is rather general and can be used for other languages, as well, it is primarily oriented toward SDL and 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 $Val : Var \to D$. We assume standard data domains such as $\mathbb{N}$, $Bool$, etc. We write D when leaving the data-domain unspecified, and silently assume all expressions to be well-typed. $\Sigma = Loc \times Val$ 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 changes configurations specified by 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* to a process P' a signal s together with a value described by an expression, and (3) *assignments*. In SDL, each transition starts with an input action, hence we assume the inputs to be unguarded, while output and assignment 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 $\alpha, \alpha' \ldots$ when leaving the class of actions unspecified. For an edge $(l, \alpha, \hat{l}) \in Edg$, we write more suggestively $l \longrightarrow_{\alpha} \hat{l}$.

We define the set of internal signals Sig_{int} as the set of all signals sent by the processes of the system. Signals coming from the environment form the set of external signals Sig_{ext}. Note that it can be the case that the same signal can come both from the environment and from a process of the system.

Time aspects of a system behaviour are specified by actions dealing with *timers*. The timed semantics we use is the one described in [6, 2], and is also implemented in the DTSpin model checker [1, 4]. 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 [2].

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 value given by a natural number. A timer can be either *set* to a value $on(v)$, i.e., it is activated to run for the designated period v, or deactivated, 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, i.e., if $[\![t]\!]_\eta = on(0)$, where $[\![t]\!]_\eta$ denotes the evaluation of the timer t in the valuation η. Both timeout and reset actions lead to the deactivation of the timer.

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 by a triple of process (of destination/origin resp.), signal, and value being transmitted.

An input of a signal, $l \longrightarrow_{?s(x)} \hat{l} \in Edg$, is enabled if the signal at the head of the queue matches signal expected by the process. Inputting results in removing the signal $s(v)$ from the head of the queue and updating the local valuation $\eta_{[x \mapsto v]}$ according to parameters of the signal. Discard is a specific feature of SDL: 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. A possible discard of a timeout signal is imitated by the deactivation of an expired timer.

Output, $l \longrightarrow_{g \,\triangleright\, P'!(s,e)} \hat{l} \in Edg$, is guarded, so sending a message involves evaluating the guard and the expression according to the current valuation. It leads to change of location of the process that sends the message. In the output edge, P' stands for the process identity of the destination. Receiving a message by asynchronous communication simply means putting it into the input queue of the process matching the identity of the signal destination. Assignments, $l \longrightarrow_{g \,\triangleright\, x:=e} \hat{l} \in Edg$, results in the change of location and the update of the valuation $\eta_{[x \mapsto v]}$, where $[\![e]\!]_\eta = v$. 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 global transitions synchronize via their common labels. Asynchronous communication between the two processes uses signal s to exchange a common value v. As far as τ-steps and non-matching communication steps are concerned, each process can proceed on its own by the interleaving. Time elapses by counting down active timers till zero, when no other actions are possible except for receiving messages from the environment. That happens only if input queues of all processes are empty and no timeout is possible.

3. Abstracting data

Originating from a chaotic environment, signals from outside can carry any value, which renders the system infinite state. Assuming nothing about the data means one can conceptually abstract values from outside into one abstract value, which basically means to ignore these data and focus on the control structure. Beside that, data not coming from outside is left untouched.

To keep the implementation in Spin's input-language Promela simple, the abstraction is realized as a straightforward source code transformation. We describe the abstraction here in terms of the abstract language from above; examples in concrete Promela-code are shown in Section 5. Instead of extending the data domains by one single additional abstract value for external data, each variable x has associated a boolean flag b_x to remember, whether its current value is a value from outside or not: The flag's value is $false$ when x contains data from outside, and $true$ otherwise. Expressions are interpreted strictly with respect to chaotic data and we write $b(e)$, where $b(e)$ is $true$ iff all of the variables occurring in e have their flags set to $true$.

As the abstract system must show at least all behavior of the original system, actions with guards whose result depends on values coming from outside, i.e. guards g with $b(g) = false$, must be enabled. Therefore we replace each guard by a transformed guard $g^\sharp$ given by the disjunction $\neg b(g) \vee g$. Timer guards are left unchanged. To propagate the information through the system, the parameter lists of signals exchanged within the system, i.e., signals from Sig_{int}, are extended with the lists of corresponding flags. Rules INPUT_{int} and OUTPUT_{int} in Table 1 show transformations for the case of a signal with one parameter.

Inputs from the chaotic environment are always enabled. We must make sure, however, that inputs from the environment do not prevent time progress. Therefore, as in [13], we

$$\frac{l \longrightarrow_{?s(x)} \hat{l} \in Edg \qquad s \in Sig_{int}}{l \longrightarrow_{?s(x,b_x)} \hat{l} \in Edg^\sharp}\ \textsc{Input}_{int}$$

$$\frac{l \longrightarrow_{?s(x)} \hat{l} \in Edg \qquad s \in Sig_{ext}}{l \longrightarrow_{g_{t_P}\, \triangleright\, set\ t_P:=0} \longrightarrow_{b_x:=false} \hat{l} \in Edg^\sharp}\ \textsc{Input}_{ext}$$

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

$$\frac{l \longrightarrow_{g\, \triangleright\, P'!(s,e)} \hat{l} \in Edg}{l \longrightarrow_{g^\sharp\, \triangleright\, P'!(s,e,b(e))} \hat{l} \in Edg^\sharp}\ \textsc{Output}_{int}$$

$$\frac{l \longrightarrow_{g\, \triangleright\, P_{env}!(s,e)} \hat{l} \in Edg}{l \longrightarrow_{skip} \hat{l} \in Edg^\sharp}\ \textsc{Output}_{ext}$$

$$\frac{l \longrightarrow_{g\, \triangleright\, x:=e} \hat{l} \in Edg}{l \longrightarrow_{g^\sharp \wedge b(e)\, \triangleright\, x:=e} \longrightarrow_{b_x:=true} \hat{l} \in Edg^\sharp}\ \textsc{Assign}_1$$

$$\frac{l \longrightarrow_{g\, \triangleright\, x:=e} \hat{l} \in Edg}{l \longrightarrow_{g^\sharp \wedge \neg b(e)\, \triangleright\, b_x:=false} \hat{l} \in Edg^\sharp}\ \textsc{Assign}_2$$

$$\frac{l \longrightarrow_{g\, \triangleright\, set\ t:=e} \hat{l} \in Edg}{l \longrightarrow_{g^\sharp \wedge b(e)\, \triangleright\, set\ t:=e} \longrightarrow_{b_t:=true} \hat{l} \in Edg^\sharp}\ \textsc{Set}_1$$

$$\frac{l \longrightarrow_{g\, \triangleright\, set\ t:=e} \hat{l} \in Edg}{l \longrightarrow_{g^\sharp \wedge \neg b(e)\, \triangleright\, set\ t:=0} \longrightarrow_{b_t:=false} \hat{l} \in Edg^\sharp}\ \textsc{Set}_2$$

$$\frac{l \longrightarrow_{g\, \triangleright\, reset\ t} \hat{l} \in Edg}{l \longrightarrow_{g^\sharp\, \triangleright\, reset\ t} \longrightarrow_{b_t:=true} \hat{l} \in Edg^\sharp}\ \textsc{Reset}$$

$$\frac{l \longrightarrow_{g_t\, \triangleright\, reset\ t} \hat{l} \in Edg}{l \longrightarrow_{g_t^\sharp\, \triangleright\, reset\ t} \longrightarrow_{b_t:=true} \hat{l} \in Edg^\sharp}\ \textsc{Timeout}$$

$$\frac{}{l \longrightarrow_{(g_t^\sharp \wedge \neg b_t)\, \triangleright\, set\ t:=1}\ l \in Edg^\sharp}\ \textsc{NoTimeout}$$

Table 1. Transformation rules

add a new timer variable t_P for each process, used to guard inputs from outside and assure time progress (cf. $\textsc{Input}_{ext}$). This timer is set to 0 until NOINPUT step is taken non-deterministically, which sets the timer to 1, thereby postponing the possibility of taking the next input from the environment until time progresses. Flags of variables which received their values from environment signals are evaluated to *false* to indicate that from this point the value is not reliable any more. Like inputs, outputs sent to a process within the system have their parameter lists extended with the corresponding flags; outputs to the environment are just removed (cf. rules $\textsc{Output}_{int}$ and $\textsc{Output}_{ext}$).

Assignments are performed only if the assigned expression is not influenced by the environment; the corresponding flag is then set to indicate that the value of the variable is reliable from now on (cf. rule $\textsc{Assign}_1$). The assignment of an unreliable value is skipped, with setting the corresponding flag to *false* to show that the value of the variable became chaotic (see rule $\textsc{Assign}_2$).

For the implementation, we use an idea similar to the one from [3], where an optimization based on *live variable analysis* is described, namely resetting the variables which are not used afterwards to its respective default value, for instance in case of integers to 0. While being simple, this transformation proved to be very efficient for the reduction of the state space in [3]. In our case, values of chaotic variables are irrelevant until the variable gets a concrete value again, so they can be reset to a default value when the flag of the variables is changed from *true* to *false*. For the sake of simplicity, we do not include this transformation into the rules of Table 1.

Concerning *timers*, the set operation and its transformation is similar to an assignment. If the expression e in $set\ t := e$ is non-chaotic, the timer is set and t's flag b_t gets the value *true* (see rule $\textsc{Set}_1$). If otherwise the expression is chaotic (cf. rule $\textsc{Set}_2$), we set the timer to 0 since in the abstraction, a chaotic timer must be able to expire immediately; the flag of the timer is set to *false*.

Resetting a timer, the timer variable gets the concrete value *off*, independent of its previous value. So the action stays unchanged while the flag of the timer gets the *true* value (cf. rule RESET). The same happens with a timeout of the non-chaotic timer (cf. rule TIMEOUT). According to this rule the same actions can be taken for the chaotic timer as well, i.e., it can expire immediately. The expiration of the chaotic timer can, however, be postponed according to rule NOTIMEOUT by non-deterministically setting the timer to 1 at an arbitrary moment in time.

4. Optimizing the transformation

For model checking, memory and time consumption are crucial. The transformation we described in the previous section is not optimal in that regard. For instance, a system can contain variables never influenced by the environment, hence we need no boolean flags for these variables. Our task is thus to localize the redundancy introduced with the transformation and eliminate it prior to verification.

The optimization of this section removes the flags whose value at every location can be determined by static dataflow analysis. In every action, the value of a flag calculated at the corresponding location can be simply substituted for the flag itself. We chose to perform the analysis on the original system, not on the transformed one, and incorporate the analysis results into the transformation.

For the optimization presented below, we need to know for each variable including the timers in each location, whether

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

For the analysis, we abstract each data domain into a two-valued domain $\{\bot, \top\}$ where $\bot$ stands for a reliable data not influenced from the environment and $\top$ for unreliable or chaotic data. Correspondingly, timers can take abstract values from $\{\mathit{off}, \mathit{on}(\bot), \mathit{on}(\top)\}$, i.e., even if the actual timer value may be chaotic, we insist on distinguishing active from non-active timers. Correspondingly, abstract valuations η^α map variables to the set of abstract values $\mathit{Val}^\alpha = \mathit{Var} \rightarrow \{\bot, \top\}$, and the analysis gives back an abstract valuation for each program location. The abstract values are ordered $\bot \leq \top$, and with this ordering, the set of valuations forms a complete lattice. We write $\eta_\bot$ for the least element, given as $\eta_\bot(x) = \bot$ for all $x \in \mathit{Var}$, and dually $\eta_\top$ for the greatest element. Furthermore, 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). Dually, we write $\bigwedge_{i=1}^n \eta_i^\alpha$ and $\eta_1^\alpha \wedge \eta_2^\alpha$ for the greatest lower bounds.

Depending on whether we are interested in an answer to point 1 or 2 from above, $\top$ is interpreted as a variable guaranteed to be influenced from outside in the first case, and dually for the second case $\top$ stands for potentially influenced from outside.

We present here only case 1, the second one is almost dual.[2] For each process of the system we construct a control flow-graph whose nodes correspond to the actions of the process. A transfer function, describing the change of the abstract valuation depending on the action at the node, is defined in Table 2. The abstract valuation $[\![e]\!]_{\eta^\alpha}$ for an expression e equals $\bot$ iff all variables in e are evaluated to $\bot$, $[\![e]\!]_{\eta^\alpha}$ is $\top$ iff the abstract valuation of at least one of the variables in e is $\top$. For inputs, $?s(x)$ in process P assigns $\bot$ to x if the signal is sent to P with reliable data, only. This means the values after reception correspond to the greatest lower bound over all expressions which can occur in a matching send-action. In the clause for input, φ expands to the condition "$\alpha_{n'} = g \rhd P!s(e)$ for some node n'", denoting all expressions in corresponding sending actions. All other clauses from Table 2 are straightforward.

For each node n of the flow graph, the data-flow problem is specified by two inequations or constraints (1). 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 2. The greatest fixpoint of the constraint set can be solved iteratively in a fairly standard way by a *worklist algorithm* (see e.g., [9, 5, 10]), where the worklist steers the iterative loop until the greatest fixpoint is reached (cf. Figure 1).

[2] The exact duality breaks for the start value of the worklist. Details can be found in [8].

$$
\begin{array}{rcl}
f(?s(x))\eta^\alpha & = & \begin{cases} \eta^\alpha[x \mapsto \top] & s \in \mathit{Sig}_{ext} \\ \eta^\alpha[x \mapsto \bigwedge\{[\![e]\!]_{\eta^\alpha} \mid \varphi] \end{cases} \\
f(g \rhd P!s(e))\eta^\alpha & = & \eta^\alpha \\
f(g \rhd x := e)\eta^\alpha & = & \eta^\alpha[x \mapsto [\![e]\!]_{\eta^\alpha}] \\
f(g \rhd \mathit{set}\ t := e)\eta^\alpha & = & \eta^\alpha[t \mapsto \mathit{on}([\![e]\!]_{\eta^\alpha})] \\
f(g \rhd \mathit{reset}\ t)\eta^\alpha & = & \eta^\alpha[t \mapsto \mathit{off}] \\
f(g_t \rhd \mathit{reset}\ t)\eta^\alpha & = & \eta^\alpha[t \mapsto \mathit{off}]
\end{array}
$$

Table 2. Abstract effect for process P

---

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

$\forall n.\ \eta^\alpha(n) = \eta^\alpha_{init}(n)$;
$\mathit{WL} = \{n \mid \alpha_n = (g \rhd x := e)\}$;

repeat
  pick $n \in \mathit{WL}$;
  let $S = \{n' \in \mathit{succ}(n) \mid f_n(\eta^\alpha(n) \not\geq \eta^\alpha(n')\}$
  in
    for all $n' \in S$: $\eta^\alpha(n') := \bigvee\{f(\eta^\alpha(n)), \eta^\alpha(n')\}$;
  if $n = g \rhd P!s(e)$
  then let $S' = \{n' \in P \mid n' = ?s(x), \eta^\alpha(n)(e) \not\geq \eta^\alpha(n')(x)\}$
  $\mathit{WL} := \mathit{WL} \backslash n \cup S \cup S'$;
until $\mathit{WL} = \emptyset$;

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

Figure 1. Worklist algorithm (must)

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

The worklist algorithm starts from the greatest valuation for all nodes. Initially, the worklist WL includes only nodes corresponding to assignment actions, since these are known to generate definite, non-chaotic values. The algorithm makes the valuation smaller step by step, until it stabilizes, i.e., until the worklist is empty. At every step we pick up some node n from the list, calculate for it the new post-valuation based on its current pre-valuation and check whether the change of the post-valuation can influence pre-valuations of the successors of n. After termination the algorithm yields two mappings $\eta^\alpha_{pre}, \eta^\alpha_{post} : \mathit{Node} \rightarrow \mathit{Val}^\alpha$. Since the lattice of valuations is finite, termination of the algorithm is trivial. On a location l, the result of the analysis

$$\frac{l \longrightarrow_{g \,\triangleright\, x:=e} \hat{l} \in Edg^{\top} \quad x \notin Var_{\text{⌶}} \quad [\![x]\!]_{\eta_l^\alpha} = \top}{l \longrightarrow_{g^\sharp \,\triangleright\, skip} \hat{l} \in Edg^\sharp} \textsc{Assign}_1$$

$$\frac{l \longrightarrow_{?s(x)} \hat{l} \in Edg^{\top} \quad x \notin Var_{\text{⌶}} \quad s \in Sig_{int}}{l \longrightarrow_{?s(x,\_)} \hat{l} \in Edg^\sharp} \textsc{Input}^{\text{⌶}}_{int}$$

$$\frac{l \longrightarrow_{?s(x)} \hat{l} \in Edg^{\top} \quad x \notin Var_{\text{⌶}} \quad s \in Sig_{ext}}{l \longrightarrow_{g_{t_P} \,\triangleright\, set\ t_P:=0} \hat{l} \in Edg^\sharp} \textsc{Input}^{\text{⌶}}_{ext}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, set\ t:=e} \hat{l} \in Edg^{\top} \quad t \notin Var_{\text{⌶}} \quad [\![t]\!]_{\eta_l^\alpha} = \top}{l \longrightarrow_{g^\sharp \,\triangleright\, set\ t:=0} \hat{l} \in Edg^\sharp} \textsc{Set}_1$$

$$\frac{t \notin Var_{\text{⌶}} \quad [\![t]\!]_{\eta_l^\alpha} = \top}{l \longrightarrow_{g_t^\sharp \,\triangleright\, set\ t:=1} \in Edg^\sharp} \textsc{NoTimeout}_1$$

Table 3. Optimized transformation for P

is given by $\eta^\alpha(l) = \eta^\alpha_{post}(n)$, also written as η^α_l.

The algorithm for point 2 calculates the least fixpoint and works in a dual way, where we start initially with a worklist that contains all nodes with input from the environment.

We use now the results of the two analyses in order to optimize the transformation of the system. Overloading the symbols $\top$ and $\bot$ we mean for the rest of the paper: the value of $\top$ for a variable at a location refers to the result of the must analysis from point 1, i.e., the definite knowledge that the data is chaotic for all runs. Dually, $\bot$ stands for the definite knowledge of the may-analysis from case 2, i.e., for data which is never influenced from outside. Additionally, we write ⌶ in case neither analysis gave a definite answer.

Clearly, the flags introduced in Section 3 are needed only for variables and timers that carry the value ⌶ (or respectively $on(\text{⌶})$ for timers) in at least one location; for other variables the values found with the static analysis can be used. So let $Var_{\text{⌶}}$ be the set of variables and timers that carry the value ⌶, respectively $on(\text{⌶})$, at least once. The rules of Table 1 are used unchanged only for variables and timers from $Var_{\text{⌶}}$. In case $x \notin Var_{\text{⌶}}$, the transformation is performed according to rules from Table 3. So an assignment of chaotic value to a variable x such that $x \notin Var_{\text{⌶}}$ is skipped (cf. rule $\textsc{Assign}_1$). In inputs, the boolean parameter for such variables is skipped, as well (cf. rule $\textsc{Input}^{\text{⌶}}_{int}$). $\textsc{Input}^{\text{⌶}}_{ext}$ covers the similar case for signals from the environment. Rule $\textsc{Set}_1$ transforms the setting of a timer marked $\top$ into setting it to the abstract value $on(0)$. $\textsc{NoTimeout}_1$ is used for arbitrary postponing a timeout of timers influenced by the environment. Timeout and reset actions are left unchanged for variables that never become ⌶. The actions on variables, that are never ⌶ and marked as $\bot$ are left unmodified, as well. The transformation of the guards is optimized in that the guards, which contains at least one variable marked $\top$, are transformed to $true$. If all the variables of a guard are marked as $\bot$, the guard is left unchanged.

5. Examples

In this section we present a simple illustrative example showing the difference between the approach of [13, 8] and the one of this paper. Unlike the technical development in the previous sections, the examples are given concretely in DTPromela, the input language of DTSpin [4]. The former approach pessimistically removes all data potentially influenced by data from outside and the transformation is based on a *static* may analysis. The latter treats data from outside *dynamically,* thus achieving a greater precision, but removes parts afterwards which are guaranteed to be chaotic, as given by the combined analysis of Section 4.

The difference is visible at locations, where the abstract valuation of some variable can get both $\top$ and $\bot$ depending on the system run. In the static approach of [13, 8], the variable instance at this location is handled as chaotic independently of the run; now the value of the variable is treated according to the value of its associated boolean flag, as described in Section 3. The simplest situation of this sort is when the variable gets its value from a signal that can be received both from the environment, and from another process of the system with a reliable value.

```
proctype A{
  ...
  pa: atomic{
    if :: chA?a,x -> goto decision; fi;
  };
  decision: atomic{
    if
    :: (x==0) -> chB!c; goto pa;
    :: (x==1) -> chEnv!c; goto pa;
    fi }
  ...
}

proctype B{
  ...
  start: atomic{
    set(tB, 5); goto wait_tB;}
    wait_tB:atomic{
      if
      :: (n>0 && expire(tB)) -> chA!a(0);
         n = n-1; set(tB,5); goto wait_tB;
      :: chB?c -> set(tB, 1); goto wait_tB;
      fi }
}
```

Figure 2. Example

```
proctype Env{
  ...
  pe: atomic{
    if
    :: expire(t) -> set(t, 1);
      n=BUFFSIZE; goto pe;
    :: (n>0 && expire(t)) ->; chA!a, 1;
      n = n-1; set(t, 0); goto pe;
    :: (n>0 && expire(t)) -> chA!a, 0;
      n = n-1; set(t, 0); goto pe;
    :: proch?c -> goto pe;
    fi ;
  }
}
```

Figure 3. Environment

As illustration, we take two processes communicating with each other and with the environment. Figure 2 shows a part of the Promela code of the system specification. Process `A` can receive signal `a(x)` both from process `B` and from the environment. Moreover, `B` always sends this signal with a concrete value.

The Promela code of the chaotic environment given as external process is shown in Figure 3. The queues in DTSpin are bounded, so we use variable `n` to limit the number of message that process `B` and the environment can send to `A` during one time slice. Otherwise, the system would deadlock in the attempt of `A` to send a message to the full queue of the environment while the environment is trying to send a message to the full queue of the process.

Furthermore, the environment must behave chaotically also wrt. the timing behavior. Therefore, sending actions of the environment are guarded by a timeout, which allows to postpone sendings until the next time slice.

Figure 4 shows the result of closing by *embedding* chaos based on the static approach from [8] and Figure 5 shows the result obtained with the method described in this paper. The old may-analysis marks variable `x` in process `A` as $\top$; therefore, the guards `x==1` and `x==0` are transformed to $true$. As a consequence, the property of the original system, that for every request `a` sent by process `B` to process `A`, process `B` eventually gets an answer `c` from `A` does not hold anymore, since `A` can send the answer to the environment instead. According to the approach of this paper, we do not take the pessimistic view but follow the information about the reliability of the value of `x` dynamically during the system run. Therefore, `B` always gets an answer from `A` for every its request sent. Thus the false negative that is obtained during the model checking in the first case does not appear when we model check the closed system in the second case.

All three variants of closing sketched here were model-

```
proctype A{
  ...
  pa: atomic{
    if
    :: expire(tC) -> set(tC, 0);
      goto decision;
    :: chA?a,x -> goto decision;
    :: expire(tC) -> set(tC, 1); goto pa;
    fi; };
  decision: atomic{
    if
    :: chB!c; goto pa;
    :: goto pa;
    fi; }
}

proctype B{
  ...
  :: expire(tB) -> pAch!a(0);
     set(tB,5); goto wait_tB;
  ...
  }
```

Figure 4. Transformed system (*may*-analysis)

checked with DTSpin. The results of the experiments confirm that closing the system with the approach of [8] and by the one presented here allow to reduce time and memory consumption compared with the system closed by adding the environment as a process.[3] So far we tested only the previous approaches on a realistic example, a larger part of a protocol, but not the enhancement developed here, for which we exercised on smaller toy examples. The results reported in [8, 14], and earlier in [12], where the results were achieved by closing and abstracting subsystems manually, show that the rather general abstraction of the environment into chaos can nevertheless be useful in praxis to help debugging large software systems by model checking. Based on first experiments on smaller examples, we except further reduction of the state space and we are currently extending the implementation to work with the extension presented here.

6. Conclusion

Model checking has gained popularity in industry and is becoming a constituent part of software engineering practice since it is, in principle, a push-button verification technology. The further dissemination of model checking, however, depends on whether it is possible to reduce the significant human involvement in applying the concomitant techniques like abstraction; automation of these techniques is therefore crucial.

[3] See [8] for comparison "embedded environment vs. environment as an external process".

```
proctype A{
  ...
  pa: atomic{
    if
    :: chA?a,x, bx -> goto decision;
    :: expire(tC) -> set(tC, 1); goto pa;
    :: expire(tC) ->
       set(tC, 0); bx=false; goto decision;
    fi; };
  decision: atomic{
    if
    :: ((x==0&bx) || (!bx)) -> pBch!c;
       goto pa;
    :: ((x==1&bx) || (!bx)) -> goto pa;
    fi }
}

proctype B{
  ...
  :: expire(tB) -> chA!a(0, true);
     set(tB,5); goto wait_tB;
  ...
}
```

Figure 5. Transformed system (*must*-analysis)

In this paper we proposed an approach for *automatic* closing of open systems, based on data and control abstraction of the environment, taking the most general environment, i.e., the chaotic one. To avoid the detrimental effect of external queues on the state space, the closing environment is embedded into the system. The approach presented here goes beyond [13] in yielding a more refined abstraction. The price for the refinement is a possible (but not necessary) increase of the state space, though the state space of the model is still significantly smaller than the state space of the model closed with the environment built as an outside chaotic process. We partially remove the additional state space without losing precision by an a-posteriori static analysis, determining variable occurrences that are guaranteed not to be influenced from outside and those which are guaranteed to be chaotic.

Procedures Since SDL and Promela feature procedures, we are currently extending our approach to handle them. A procedure with formal parameters can be invoked by different processes of the system both with chaotic and non-chaotic values of parameters. In the presented closing approach, the transformation of procedures is rather straightforward and can be handled in analogy to communication: A procedure call is treated like a special signal and the parameter lists of procedures get thus extended with boolean flags to work with abstract data, as presented in Section 3.

Future work For future work, we will extend our tool for closing open components with the algorithm described here. We also need to extend the method to account for more complex data types and process creation. Based on the results from [15], another direction for future work is to extend the PML2PML implementation to handle environments more refined than just chaos with building an environment process communicating to the system synchronously.

References

[1] 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.

[2] 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.

[3] M. Bozga, J. C. Fernandez, and L. Ghirvu. State space reduction based on Live. In A. Cortesi and G. Filé, editors, *Proceedings of SAS '99*, volume 1694 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[4] Discrete-time Spin. `http://win.tue.nl/~dragan/DTSpin.html`, 2000.

[5] M. S. Hecht. *Flow Analysis of Programs*. North-Holland, 1977.

[6] 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.

[7] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[8] N. Ioustinova, N. Sidorova, and M. Steffen. Closing open SDL-systems for model checking with DT Spin. In L.-H. Eriksson and P.-A. Lindsay, editors, *Proceedings of Formal Methods Europe (FME'02)*, volume 2391 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[9] G. Kildall. A unified approach to global program optimization. In *Proceedings of POPL '73*, pages 194–206. ACM, January 1973.

[10] F. Nielson, H.-R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[11] Specification and Description Language SDL, blue book. CCITT Recommendation Z.100, 1992.

[12] 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.

[13] 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.

[14] 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.

[15] 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.