Divide, Abstract, and Model-Check*

Karsten Stahl**, Kai Baukus, Yassine Lakhnech, and Martin Steffen

Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität zu Kiel
Preußerstr. 1–9, D-24105 Kiel, Germany
{kst,kba,yl,ms}@informatik.uni-kiel.de

Abstract. The applicability of model-checking is often restricted by the size of the considered system. To overcome this limitation, a number of techniques have been investigated. Prominent among these are data independence, abstraction, and compositionality. This paper presents a methodology based on deductive reasoning and model-checking which combines these techniques. As we show, the combination of abstraction and compositionality gives a significant added value to each of them in isolation. We substantiate the approach proving safety of a sliding window protocol of window size 16 using Spin and PVS.

1 Introduction

Model-checking [CE81,QS81] has proven a valuable approach for the formal, automatic verification of reactive systems. The size of the system to be verified limits, however, the applicability of this approach. First of all, many applications such as protocols use *infinite* data domains. This immediately renders the state space infinite, and hence, simple state exploration fails. Even when dealing with finite data, systems of parallel processes yield a state space exponential in the number of processes. This is known as the *state explosion* problem.

The obstacle of infinite data domains can be tackled by the *data independence* technique [Wol86]. Intuitively, a program is data independent if its behavior does not depend on the specific values of the data. In this case, many properties of the program stated over an infinite data domain can be equivalently expressed over finite domains that must contain enough elements. We call *safety* of data independence the requirement that the finite and infinite properties are equivalent.

Abstraction techniques are a prominent approach to address the state explosion problem (see e.g. [CGL94,BBLS92,Lon93,DGG94,Dam96]). Abstraction is a general approach [CC77] which allows to deduce properties of a concrete system by examining a more abstract —and in general smaller— one. Both systems are connected by an abstraction relation which is called *safe* with respect to a given property, if it preserves satisfaction of the property. This means, whenever the

* This work has been partially supported by the Esprit-LTR project Vires.
** Contact author.

property holds for the abstract system, it holds for the concrete one as well. In general, for a given concrete system, the abstract one depends on the property to be established. Therefore, in case the property to be verified is *global,* i.e., it depends on the exact behavior of all processes, it is hard to significantly abstract these components. It should be intuitively clear that the more local to a set of processes a property is, the more radically the remaining processes can be abstracted. Therefore, it is appealing to decompose a global property into a number of *local* ones which together imply the original one. Breaking a verification problem into more manageable subproblems is the essential idea of *compositional reasoning* (see e.g. [dRLP98] for a recent collection of relevant work in this field). To summarize, combining abstraction techniques and compositionality gives a significant added value to each of these techniques in isolation.

Verification of safety of data independence, of safety of abstraction, and of decomposition correctness is, in general, an undecidable problem that can be tackled using *deductive reasoning.* This is where theorem-proving comes in our approach.

The contribution of this paper is to explore the proposed methodology and to substantiate its applicability with a safety proof of a *sliding-window* protocol. The chosen variant of the protocol is inspired by the one implemented in *Mascara* [DPA$^+$98], a medium-access layer for wireless ATM-networks, which uses a window size of 16. Judging from experience, model-checking directly the protocol with this window size is far beyond the reach of Spin or similar model-checkers. Using decomposition together with abstraction and data independence we were able to automatically verify the decomposed subproblems with the Spin model-checker [Hol91]. To verify the safety of the abstraction we used the theorem-prover PVS [ORSvH95].

The remainder of the paper is organized as follows: Section 2 explains the techniques used during verification. Section 3 contains the Promela model of the sliding window protocol. In Section 4 we present safety proof for the protocol. Finally, Section 5 contains concluding remarks and references to related work. The complete Promela code and the PVS derivation can be found at `http://www.informatik.uni-kiel.de/~kst/sw/`.

2 Verification Approach

The goal is to prove that a system $\mathcal{S}$, given as a parallel composition of a number of subsystems S_i, satisfies a given property φ, that is

$$\mathcal{S}_1 \| \dots \| \mathcal{S}_n \models \varphi \ .$$

In practice, when trying to apply model-checking in this setting, several problems occur. First, the data part is often infinite or at least very large. Second, the parallel composition of the S_i's leads to an exponential blowup of the state space of the overall system, which is known as the *state explosion problem.*

This paper presents a practical method to build abstract systems which can be handled by model-checkers. It is applied to a sliding window protocol taken from a wireless ATM protocol.

The methodology is based on three principles:

1. Decomposition of the property to prove,
2. data independence of the system, and
3. building abstractions.

2.1 Decomposition

We first *decompose* the property φ into a set of properties $\varphi_1, \ldots, \varphi_k$ which together imply φ, i.e., $\varphi_1 \wedge \ldots \wedge \varphi_k \implies \varphi$, such that each property φ_i is easier to establish than the original property φ. In order to derive φ_i, one can in turn use the principles of *data independence*, *abstraction*, and *decomposition*

As a guideline of a decomposition one should always try to introduce properties φ_i for which only a few processes are relevant, and which are therefore *local* to these processes. In such a case, this property can be shown with very abstract versions of the remaining processes.

2.2 Data Independence

For a property φ, we make use of the *data independence* [Wol86] of a system to change the input language, favorably reducing the alphabet of the input language to a finite set.

Data independence means that the system does not change the data received nor that it invents new data. For example, the system is allowed to compose the input to build new blocks in case it is later decomposed without change, or store some data and use it later unchanged. These assumptions can be checked syntactically and often hold for data-transmission protocols, and in particular, for the sliding window protocol of the following section.

When changing the input language, usually also the property φ has to be adapted yielding a property $\widetilde{\varphi}$ over the new input language. The requirement $\widetilde{\varphi}$ has to satisfy is that it holds for the program operating on the new input language if and only the original property holds for the program with the original input language.

For example, suppose we want to check that a process, given as input the increasing sequence of natural numbers starting from 0, first delivers a 0 as output. Since satisfaction of the property depends on whether 0 or *any other* value appears first, one can identify all values other than 0. In other words, under the assumption that the process is data independent, one can show the stated property with the restricted input language $0\,1^\infty$, using only a finite alphabet instead of the natural numbers.

2.3 Abstraction

Abstraction [CC77,CGL94,Lon93,DGG94,Kur94,LGS+95,Dam96,Kel95] is the third technique we use. It is a general way of deriving properties for systems by investigating smaller, more abstract ones. In our specific setting, to establish a property φ local to component $\mathcal{S}_j$, we create abstract versions $\widetilde{\mathcal{S}_l}$ of the remaining processes $\mathcal{S}_l$ such that

$$\widetilde{\mathcal{S}_1}\| \dots \|\widetilde{\mathcal{S}_{j-1}}\|\mathcal{S}_j\|\widetilde{\mathcal{S}_{j+1}}\|\widetilde{\mathcal{S}_n} \models \varphi \quad \text{implies} \quad \mathcal{S}_1\| \dots \|\mathcal{S}_n \models \varphi \ .$$

Of course, in general the part of interest might not consist of a single process $\mathcal{S}_j$ but of a set of processes. The connection between the concrete and the abstract system is given by an abstraction relation α between the two state spaces. Since we consider only *path universally quantified* properties, it is sufficient that the abstract versions $\widetilde{\mathcal{S}_l}$ exhibit more behavior than the original $\mathcal{S}_l$ with respect to the abstraction relation, i.e., the abstraction relation is a *simulation* relation. Formally, α is a simulation relation between both systems, if the following condition, called *safety of abstraction*, holds:

$$\forall c, c' \in \Sigma_C, a \in \Sigma_A . \tau_C(c, c') \wedge \alpha(c, a) \implies \exists a' \in \Sigma_A . \tau_A(a, a') \wedge \alpha(c', a') \ ,$$

where Σ_C is the concrete state space and τ_C is a concrete transition (respectively Σ_A and τ_A for the abstract system).

In the verification of the sliding window protocol, we prove safety of abstraction using the theorem prover PVS [ORSvH95]. The translation of the system transitions and the state space into a PVS theory is straightforward and omitted from the paper.

2.4 Verification Strategies

The techniques presented above can be applied in any suitable order and the application can be iterated. Having in mind that a property should be decomposed into more local properties, it is, however, advantageous to first apply decomposition. Indeed, it does not enlarge the state space but rather gives more possibilities for applying the two other abstraction techniques.

In the sequel, we present an iterative method which can be applied in order to decompose a given property into more local ones. To do so, assume we are given two processes S_1 and S_2 in parallel, and a property φ. Suppose we want to show $S_1 \parallel S_2 \models \varphi$ and let C denote the chaotic process which exhibits all possible behaviors. In Fig. 1, we describe an (semi-)algorithm given in pseudocode which can be applied to decompose a property into more local properties. In the given description we tacitly identify processes and properties and denote by $\mathcal{R}(S)$ the set of reachable states of S.

Clearly, we can replace $\mathcal{R}(S_1 \parallel \psi_2)$ (resp. $\mathcal{R}(\psi_1 \parallel S_2)$) by any property which follows from $\mathcal{R}(S_1 \parallel \psi_2)$ (resp. $\mathcal{R}(\psi_1 \parallel S_2)$) and axioms which can be derived from the semantics such as the usual assumptions concerning the buffers.

We now illustrate this methodology by proving safety of a sliding window protocol used in *Mascara* [DPA+98], a medium-access layer for wireless ATM-networks. We start with a description of the protocol.

$\psi_1 := C;$
$\psi_2 := C;$
Do
  $\psi_1 := \mathcal{R}(S_1 \| \psi_2)$
[]
  $\psi_2 := \mathcal{R}(\psi_1 \| S_2)$
Until $\psi_1 \wedge \psi_2 \Rightarrow \varphi$

Fig. 1. Property Decomposition

3 The Sliding Window Protocol

The sliding window protocol [Ste76] is a communication protocol to guarantee reliable data transmission over unreliable, buffered communication channels. Considering only unidirectional communication, the protocol consists of a sender and a receiver process, connected by two channels, one in each direction (cf. Fig. 2).

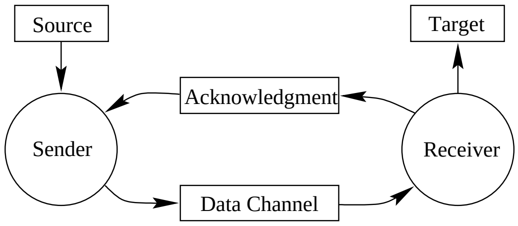

Fig. 2. Communication structure

The sender receives data from the source which has to be transmitted to the target. The data is sent to the receiver over the data channel, which may lose but not reorder messages. The sender stamps each data item with a sequence number such that the receiver can detect whether a message had been lost. The receiver acknowledges received data by sending the sequence number which it expects next over the lossy acknowledgment channel. The sender can retransmit unacknowledged messages, which are kept in the so-called *transmission window* until acknowledged. To hand out the data items to the target in the correct order, the receiver can temporarily store them in its *reception window.*

The protocol has three important parameters:

Transmission window size tw**:** This value specifies the maximal number of messages sent but not yet acknowledged together with those received from the source but not yet sent.

Reception window size rw**:** This is the maximum number of messages that can be kept at the receiver side without being delivered.

Cardinality of sequence numbers n: Since the transmission window is finite, a finite set of sequence numbers suffices to ensure unique identification of each message. The protocol works correctly with at least $n = tw + rw$ sequence numbers. In this case the numbers $\{0, \ldots, n-1\}$ are used cyclically modulo n.

Mascara's sliding window protocol uses a transmission window size of 15 and size 1 for the receiver, which means the receiver either delivers a message in case it fits into the output stream or it discards it. The protocol uses the minimal possible amount of sequence numbers, namely 16. Although in general, a sliding window protocol assures safety over unbounded lossy FIFO-channels, in Mascara the buffer is restricted to a size of 16.

In the following we give the Promela model (Promela is the input language of Spin) for sender and receiver.[1] In Sect. 4 we will refer to the transitions of sender and receiver using the names mentioned in the comments of the Promela code. The lossiness of communication is modeled using Promela communication channels of buffer size 16, where the receiving side may decide nondeterministically to lose the message.

```
proctype Sender()
{
  Data transmit_window[window_size];
  Window_index last_unacknowledged = 0;
  Window_index next_to_send        = 0;
  Window_index next_free           = 0;
  Seq_Number ack;

  do
  :: atomic{                                      /* receive_data */
     !(data_window_full) ->
       gen_to_send?(transmit_window[next_free]);
       next_free = ((next_free + 1) % window_size)
     }
  :: d_step{                                      /* send         */
     data_pending && nfull(send_to_rec) ->
       send_to_rec! transmit_window[next_to_send], next_to_send;
       next_to_send = (next_to_send + 1) % window_size
     }
  :: d_step{
     sliding_window_full ->                       /* timeout      */
       next_to_send = last_unacknowledged
     }
  :: d_step{                                      /* receive_ack  */
     nempty(rec_to_send) -> rec_to_send?ack;
       if
       :: (ack == last_unacknowledged)  ->
```

[1] In the Promela code we leave out macro definitions for the data operations, which should be clear from the context.

```
            next_to_send = last_unacknowledged
       :: else -> last_unacknowledged = ack
       fi
  :: rec_to_send?_                                /* ack_lost     */
     }
  od
}
```

As long as the transmission window is still capable of storing messages to be sent, the sender can read new data items from the data source. The new items are put in the open part of the window together with the already sent but yet unacknowledged messages. The position in the window corresponds with the sequence number given to the data items. Those messages are sent in the cyclic order of their sequence numbers (see Fig. 3) by the `send` transition.

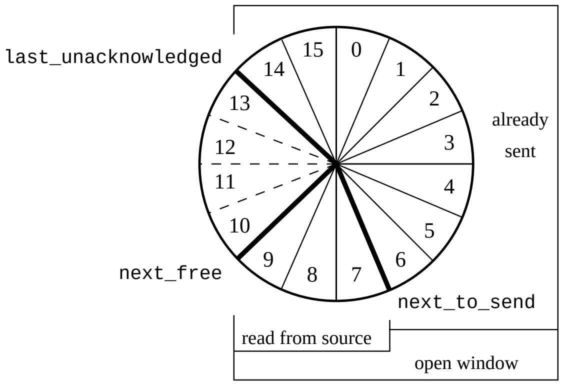

Fig. 3. Transmission window of the sender

A specific feature of Mascara's sliding window protocol is how the retransmission is triggered: in smooth operation of the protocol each acknowledgment will confirm one or more messages. The fact that twice the same acknowledgment is received is taken as indication by the sender that a transmission error has occurred and it start resending (by setting `next_to_send` back to `last_unacknowledged`). Another cause for resending in practice is a timeout occurring in case the acknowledgments are too late. This situation is modeled here in a simplified way where the timeout is bound to occur when the maximum number of 15 unacknowledged cells has been sent (`sliding_window_full`). An arriving acknowledgment causes the acknowledged message to fall out of the open window, the window 'slides' one message forward (`receive_ack`). The acknowledgment may also be lost which is modeled by `ack_lost`.

```
proctype Receiver()
```

```
{
  byte next_expected = 0;
  Seq_Number received_number;

  do
  :: send_to_rec?(_,_)                              /* receive_lost */
  :: atomic{                                        /* receive      */
     send_to_rec?(data_item,received_number) ->
       if
       :: (received_number == next_expected) ->
            rec_to_upper!data_item;
            printf("Output %d\n", data_item);
            next_expected = (next_expected + 1) % window_size
       :: else  -> skip
       fi
     }
  :: rec_to_send!next_expected                      /* send_ack     */
  od
}
```

If the receiver gets a message with the expected sequence number, it delivers the corresponding data item to the data target. The counter for the next expected sequence number is increased cyclically by 1. Also, an acknowledgment may be sent at any time, indicating the next expected sequence number (`send_ack`).

4 Verification of the Sliding Window Protocol

We want to prove a *safety property* of the sliding window protocol of Sect. 3, namely that the protocol ensures reliable communication. This means no data item is lost nor duplicated and the receiver delivers the data in the original order. Relating the input stream at the sender to the output stream delivered by the receiver, we say that the protocol is correct, if for all input sequences the output sequence of each possible run of the protocol is a *prefix* of the input sequence. Trying to establish the correctness of the protocol with the full window size of 16 directly will not succeed using Spin.

Observing that the protocol is *data independent*, we can start by considering the stream of natural numbers as input, as opposed to arbitrary data, i.e., it is enough to establish that if we have as input the stream of natural numbers, the output is always a prefix thereof.

Before we formalize the properties, we introduce some useful notations. A *sequence* of data items from a data domain D is a function $seq : \mathbb{N}_{<k} \to D$ for a $k \in \mathbb{N} \cup \{\infty\}$, this k is the *length* of *seq* and is denoted as $\#seq$. We also denote a sequence as $(seq_i)_{i<k}$, or $(seq_i)_{i\in\mathbb{N}}$ in case $\#seq = \infty$. If *out* is a possible output sequence of the sliding window protocol for a given input sequence *in*, we denote this with $out \in SW(in)$. Let $in_{\mathbb{N}} = (j)_{j\in\mathbb{N}}$. A *language* is a set of sequences. For a language L, we denote with $out \in SW(L)$ that there is a word $in \in L$

with $out \in SW(in)$. A run or a computation with a certain input is a sequence of consecutive states of the sliding window protocol.

Lemma 1. *If the protocol satisfies the property*

$$\text{For all } out \in SW(in_{\mathbb{N}}), k < \#out : out(k) = k \ , \qquad (\texttt{Prefix})$$

then it is correct in the sense described above.

Proof. Assume that the protocol is erroneous, i.e., there exists an input sequence in and a trace of the protocol such that the output sequence out is no prefix of in. Then, we find a first error on a position $k < \#out$ such that

$$out(k) \neq in(k) \text{ and for all } j < k : out(j) = in(j) \ .$$

Since the protocol is data independent, we can adapt this trace with the input $in_{\mathbb{N}}$. Data independence ensures that exactly the same steps are possible. This yields to an output sequence $\widetilde{out}$ of the same length as out. Consider now position k of $\widetilde{out}$. Then, $\widetilde{out}(k) \neq in_{\mathbb{N}}(k)$. Thus, there exists $j \neq k$ with $in_{\mathbb{N}}(j) = \widetilde{out}(k)$. But by choice of $in_{\mathbb{N}}$ we are now able to derive $\widetilde{out}(k) \neq k$.

In the following, let out be an arbitrary output sequence of the sliding window protocol for the input sequence $in_{\mathbb{N}}$.

4.1 Decomposition of (`Prefix`)

After exploiting data independence to simplify the data domain, we continue by a first decomposition step, splitting the safety requirement into the following four properties.

Lemma 2. *The following properties are a decomposition of (`Prefix`):*

$$\forall i < \#out : out(i) \in \mathcal{M}_{16}(i) \qquad (\texttt{Mod16})$$

$$\#out > 0 \implies out(0) = 0 \qquad (\texttt{Init})$$

$$\forall i < \#out - 1 : out(i+1) > out(i) - 15 \qquad (\texttt{WSize})$$

$$\forall i < \#out - 1 : out(i+1) \leq out(i) + 15 \qquad (\texttt{Lose})$$

where $\mathcal{M}_{16}(i) = \{j \in \mathbb{N} \mid j \equiv_{16} i\} = \mathbb{N} \cap i + 16\mathbb{Z}$.

The induction principle motivates this decomposition. Property (`Init`) is the induction base, saying that the first output is correct. The other properties decompose the induction step, which states that $out(i) = i \implies out(i+1) = i+1$.

One can observe that whatever happens between sender and receiver, the labeling of the input data items with the sequence numbers, and the receiving in just the right order of the sequence numbers done by the receiver, ensures Prop. (`Mod16`), restricting the positions on which input data elements can occur in the output sequence.

Properties (`WSize`) and (`Lose`) now give further restrictions to the output sequence positions, saying that the values of neighbored positions in the output sequence are not too far away.

Proof. We have to show that these properties together imply (`Prefix`), so assume they are valid. Let $i < \#out$. We prove $out(i) = i$ by induction on i.
Case: $i = 0$. In this case we have $out(0) = 0$ by (`Init`).
Case: $i \to i + 1$. Assume $out(i) = i$, $i + 1 < \#out$, and consider $out(i + 1)$. We know by (`Mod16`) that there exists a $k \in \mathbb{Z}$ such that $out(i + 1) = i + 1 + 16k$. Since $i = out(i)$ we have $out(i + 1) = out(i) + 1 + 16k$.

- If $k \geq 1$, we have $out(i + 1) \geq out(i) + 17$ in contradiction to (`Lose`).
- If $k \leq -1$, we have $out(i + 1) \leq out(i) - 15$ in contradiction to (`WSize`).

Consequently, $k = 0$ and $out(i + 1) = i + 1$.

We now establish Properties (`Mod16`), (`Init`), (`WSize`), and (`Lose`).

In the following sections we use the abbreviations R for the receiver and S for the sender process. The notion *send* means that the message has been put into the channel.

4.2 Property (`Mod16`)

First, we want to establish Prop. (`Mod16`): $\forall i < \#out : out(i) \in \mathcal{M}_{16}(i)$.

Data Independence By data independence, we can reduce the input language to the word $in_m = (0\,1\,2\,\ldots\,15)^\infty$. So we identify data items on positions $16k+i$ and $16k' + i$. We have to show the following:

(i) For all $out \in SW(in_m)$: out is a prefix of in_m.

Now we apply the decomposition principle explained in Sect. 2.4.

Considering a chaotic sender, we observe that R sends only acknowledgments n such that either $n = 0$, or $n > 0$ and R has received a message $(d, n-1)$ before. Since the messages are exchanged via a buffer, one can derive that R sends only n if either $n = 0$, or $n > 0$ and S has sent a message $(d, n - 1)$ before.

If R behaves in this way, S can only send positions of the transmit window to which it has written before. Hence, we can derive a restriction for the messages standing in the buffer. With this restriction, we can construct an abstract system usable for verification of Prop. (i).

This decomposition is explained in detail below.

Abstractions Even with the reduced input language the system is far from being model-checkable. The main problem for the verification are the buffers from S to R and vice versa. A method often successful to overcome the state explosion caused by such a buffer is to restrict the possible messages which can occur in it, and to use two variables as the abstract buffer: one which holds the head value of the buffer, and one boolean which is true if and only if the buffer is empty, since these are the important informations for one system step.

The abstraction relation then says that the abstract variable holds exactly the head value of the buffer in the case it is non-empty. This relation must be restored after each read operation by assigning arbitrary values to the abstract variables. Here, restricting these values is often helpful.

As explained, we now apply the decomposition principle from Sect. 2.4 and start considering a chaotic sender. This leads to the following property:

(ii) In every run (with input in_m), each acknowledgment n sent by R is either 0, or $n > 0$ and R received a message $(d, n-1)$ before.

Now we describe the abstraction used for the verification of this property. We abstract the buffer from S to R in the way described above, discard the acknowledgment buffer, and use a chaotic sender which is able to send any message. The receiver functionality is unchanged, except that R is not required to send, and that R changes the buffer variables arbitrarily after reading a message. Then we use Spin to establish that property.

Since every message occurring in the buffer is written to it by S, Prop. (ii) implies Prop. (iii).

(iii) In every run (with input in_m), each acknowledgment n sent by R is either 0, or $n > 0$ and S sent a message $(d, n-1)$ before.

Relying on this Prop. (iii), we obtain by setting S in parallel with a receiver fulfilling Prop. (iii):

(iv) For all messages (d, n) occurring in the buffer from S to R (at any time in any run with input in_m), $d = n$.

The abstract system used to establish Prop. (iv) by model-checking is the following. The buffer from S to R is discarded, and the acknowledgment buffer is abstracted in a similar way as before. The abstract receiver non-deterministically sends arbitrary acknowledgments allowed by Prop. (iii). The abstract sender non-deterministically assigns values allowed by Prop. (iii) to the buffer variables after an acknowledgment reception, the rest of the sender is unchanged. For this abstract system, Prop. (iv) can be model-checked.

Now we describe the abstraction used for verifying Prop. (i) with Spin. The channel from S to R is abstracted to two variables as described above, the acknowledgment buffer is discarded. After each read operation of R, R restores the abstraction relation by assigning non-deterministically a value (n, n). These are the only possible values according to Prop. (iv). The abstract sender can assign arbitrary values (restricted by Prop. (iv)) to the buffer variables. Then, the property is easily model-checked using Spin.

The abstraction relation ensures that the receiver variables of the abstract and concrete systems coincide, and that the abstract channel variable holds the head value of the concrete channel in case it is non-empty.

4.3 Property (`Init`)

Following our approach we now prove $\#out > 0 \implies out(0) = 0$.

Decomposition Decomposing (`Init`) leads to the following requirements:

(v) For all runs with input $in_{\mathbb{N}}$, if a message with a data value $d \geq 16$ is sent by S, then the sender read an acknowledgment $\neq 0$ earlier.
(vi) R gives out only data items which R has received through the data channel.

We will show (v) by model-checking. For this property we use in the following sections data independence and abstractions to yield a suitable system. Property (vi) follows easily by data independence. Indeed, we can directly derive from the program text that R does not change received data nor introduces new data, that is, it only passes on the received data items.

The decomposition leads to a proof obligation, namely that the conjunction of Prop. (v) and (vi) implies Prop. (`Init`). This is shown next.

Proof. Consider a computation, a state in that computation, and assume $\#out > 0$ in that state. With Prop. (`Mod16`) we know that $out(0) = 16k$ for a suitable $k \in \mathbb{N}$. By (v) we know that before a data item $d \geq 16$ is sent by S, the sender receives an acknowledgment a different from 0. Consider the point in the computation when S receives an acknowledgment $a \neq 0$ for the first time. Then, a was sent by R, since the buffer does not invent data.

Consequently, R has changed the value of its variable `next_expected` which is initialized to 0, and therefore, R has received a message with sequence number 0.

Every message which R receives was sent by S earlier. Since $(0, 0)$ is the only message sent by S so far with sequence number 0, R has received this message.

If R changes `next_expected`, it also gives out the data item attached to the received message in the same atomic transition. Therefore, R starts the output sequence with the data item 0.

Data Independence By data independence we reduce the input language of the system for Prop. (v) to $0^{15}\,0^*\,1\,0^\omega$. Adapting Prop. (v) to this new input language, we have to show the following:

(vii) For all runs with the input language $0^{15}\,0^*\,1\,0^\omega$, whenever S sends data value "1", S received an acknowledgment $\neq 0$ before.

Now we use abstraction to construct an abstract system used to model-check this property.

Abstraction For Prop. (vii), we can use a very abstract receiver which non-deterministically is able to send every possible acknowledgment. The functionality of the sender is unchanged, except two modifications. First, the buffer between sender and receiver can be discarded. Second, the capacity of the acknowledgment channel still leads to state space explosion. Therefore, we abstract the buffer analogously to Sect. 4.2 to two variables `rec_to_send` (for the head element) and `rec_to_send_empty`. The abstract system can set these variables to arbitrary values after reading an acknowledgment.

After constructing this abstract system, one can observe that the functionality of the sender transition `ack_lost` is almost the same than the functionality of the whole receiver. Therefore, only little modifications must be made to abstract away the whole receiver. The concrete receiver steps are then simulated by the abstract `ack_lost` transition.

Furthermore, we introduce an auxiliary variable `last_sent` which holds the value of the last data item sent and is used to formulate the property, and which replaces the abstract channel from S to R with respect to Prop. (vii).

The abstraction relation which defines the relationship between the concrete and the abstract system is given in Fig. 4. Here, σ_C (resp. σ_A) is a concrete (resp.

$$\begin{aligned}
\alpha(\sigma_C, \sigma_A) \stackrel{\text{def}}{=} \; & \sigma_C(\texttt{transmit_window}) = \sigma_A(\texttt{transmit_window}) \\
& \wedge \sigma_C(\texttt{next_to_send}) = \sigma_A(\texttt{next_to_send}) \\
& \wedge \sigma_C(\texttt{next_free}) = \sigma_A(\texttt{next_free}) \\
& \wedge \sigma_C(\texttt{last_unacknowledged}) = \sigma_A(\texttt{last_unacknowledged}) \\
& \wedge \text{nonempty}(\sigma_C(\texttt{rec_to_send})) \implies \\
& \qquad \text{head}(\sigma_C(\texttt{rec_to_send})) = \sigma_A(\texttt{rec_to_send}) \\
& \wedge \neg\sigma_A(\texttt{rec_to_send_empty}) \iff \\
& \qquad \text{nonempty}(\sigma_C(\texttt{rec_to_send})) \\
& \wedge \text{nonempty}(\sigma_C(\texttt{send_to_rec})) \implies \\
& \qquad \text{proj}_1(\text{last}(\sigma_C(\texttt{send_to_rec}))) = \sigma_A(\texttt{last_sent})
\end{aligned}$$

Fig. 4. Property Decomposition

abstract) state, head of a FIFO queue gives the next element which will be read, last gives the last value appended, proj_i is the i'th projection, and nonempty of a FIFO queue holds whenever the queue is not empty.

With this abstract system, it is possible to establish the (adapted) property using model-checking. We made the abstraction proofs using PVS. The translation of the Promela-model into a PVS theory is straightforward, and the abstraction proofs were almost automatic.

4.4 Property (`WSize`)

To establish $\forall i < \#out - 1 : out(i+1) > out(i) - 15$, we use a similar approach as for Prop. (`Init`). Therefore, we do not go into every detail for this property. The property holds, since the transmit window is bounded, and since the sender fills the transmit window in cyclic order. After a data items falls out of the window, which happens if we read too many items from the upper layer, it can never be send again.

Decomposition We only mention one of the properties we got by decomposing (`WSize`):

(viii) For all runs with input $in_{\mathbb{N}}$, if the sender sends a value d, it will never send a value $d' \leq d - 15$ afterwards.

We will establish this property by model-checking and describe in the following sections how to apply data independence and abstraction to it. The decomposition further needs the fact that we have a FIFO buffer, and that R does not store, but directly delivers when it receives data.

Data Independence To verify Prop. (viii), we use data independence to reduce the input language of the system to $L = 0^* \, 1 \, 0^{14} \, 0^* \, 2 \, 0^\infty$. Then, we have to reformulate Prop. (viii) and obtain:

(ix) For all runs with input from L, after sending data item "2", the sender never sends a "1".

Abstractions To verify the reduced property, we again concentrate on the sender and use a very abstract receiver which only sends non-deterministically an arbitrary acknowledgment. In fact, the same abstract system (apart from the input language) can be used as given in Sect. 4.3.

4.5 Property (`Lose`)

To prove $\forall i < \#out - 1 : out(i+1) \leq out(i) + 15$, let us start with the following lemma which is proven analogously to Lemma 2.

Lemma 3. *If Props. (`Mod16`)–(`Lose`) from page 9 hold for all states of a computation* $comp = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots$ *up to a state* s_n*, then* s_n *satisfies:* $\forall j \leq \#out - 1 : out(j) = j$.

For the proof of the next proposition, we introduce some new notions. A *non-empty position* of the *transmit window* in state s of the sender is a position i with $0 \leq i \leq 15$ such that

$$\begin{aligned} &(i - s(\texttt{last\_unacknowledged})) \bmod 16 \\ &< (s(\texttt{next\_free}) - s(\texttt{last\_unacknowledged})) \bmod 16 \enspace . \end{aligned}$$

In other words, it is a position in the open transmit window, one in the modulo interval $[s(\texttt{last\_unacknowledged}) \ldots \mathrm{pred}_m(s(\texttt{next\_free}))]$, where $\mathrm{pred}_m(i) = (i-1) \bmod 16$. Analogously, an *empty position* is a position which is not non-empty. The notion that the sender *acknowledges* position i by taking transition t means that, in the original state, position i was a non-empty position, and after taking transition t, i has become an empty position. The sender *acknowledges erroneously* by executing a sequence of acknowledgment receptions, when after executing the sequence steps, there are more non-empty positions than before executing that sequence. This covers the case when the pointer `last_unacknowledged` jumps to an empty position by reading an acknowledgment, thus enlarging the open transmit window.

Proposition 1. *The sliding window protocol satisfies Prop. (*`Lose`*).*

Proof. Assume that Prop. (`Lose`) does not hold. Then we find a computation

$$comp = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \dots \xrightarrow{t_{n-1}} s_n$$

such that Prop. (`Lose`) is violated. Each s_i is a state and each t_i is a transition of the system (for example a transition of the receiver). Let n be the first position such that s_n satisfies one of the following clauses:

(a) $out(i+1) > out(i) + 15$ for an i.
(b) By reading the acknowledgments from the acknowledgment buffer, S is able to acknowledge a data item d which has not been given out by R.
(c) S can acknowledge erroneously by executing a sequence of acknowledgment receptions.
(d) By reading the acknowledgments from the acknowledgment buffer, S can reach a state such that the value of `next_expected` is a bad acknowledgment in the sense of (b) or (c), i.e. S can reach a state, in which the value of `next_expected` would acknowledge a data item which has not yet been given out by R, or it would acknowledge erroneously.

It is obvious by initialization of the processes that $n > 0$.

Case: Property (a) holds in state s_n. Choose i such that (a) holds for that i. Since n is the minimal position such that (a) holds, the last transition t_{n-1} must have added $out(i+1)$ to the output sequence. Since n is the minimal position for which one of (a)–(d) holds, we know that in state s_{n-1} (a) does not hold, which implies that Prop. (`Lose`) holds together with the other Props. (`Init`)–(`WSize`).

Lemma 3 then implies, that the output sequence of s_{n-1} is $0, 1, \dots, i$. Consequently, $out(i) = i$. Step t_{n-1} extends this sequence with $out(i+1)$. By (`Mod16`) we know that $out(i+1) = i + 1 + 16k$ for a suitable k. With Prop. (a) we then derive that $k \geq 1$. Consequently, $out(i+1) > (i+1) + 16$.

Then, the sender has already sent $i+1$ before. But this data item has never been given out by the receiver. On the other hand, the sender must have acknowledged data item $i+1$ in the past, otherwise it could not get data item $out(i+1)$ into its transmit window. Hence, on an earlier point in the computation, (b) must hold. Contradiction!

Case: Property (b) holds in s_n. Hence, by reading acknowledgments from the buffer, S can acknowledge a data item d which has not yet been given out by the receiver. Consider transition t_{n-1}.

- $t_{n-1} \in \{\texttt{send}, \texttt{timeout}, \texttt{receive\_lost}, \texttt{receive}\}$: In this case, the variables `next_free`, `last_unacknowledged`, and buffer `rec_to_send` are unchanged. Hence, (b) also holds in state s_{n-1}. Contradiction!
- $t_{n-1} = \texttt{receive\_data}$: Consider first the case that the new data item received by S and inserted in the transmit window on position $j = s_{n-1}(\texttt{next\_free})$ is the one falsely acknowledged. After step t_{n-1}, the sender is able to acknowledge d by reading acknowledgments. These transitions do not affect variable `next_free`. Hence, it is unchanged, and the only possibility to remove d from the transmit window, is to set the pointer `last_unacknowledged` to position $s_n(\texttt{next\_free})$.
  Now consider that the same sequence of acknowledgment receptions is done beginning in state s_{n-1}. With this sequence of steps we reach a state s such that $s(\texttt{next\_free}) = \text{pred}_m(s(\texttt{last\_unacknowledged}))$, which means the transmit window is maximally open.
  In case the open window had this size already in s_{n-1}, it is not possible to take t_{n-1} in state s_{n-1}. Contradiction. In the other case, (c) is already satisfied in s_{n-1}.
  Now consider the case, that d was already in position j in the transmit window in state s_{n-1}. After the same sequence of acknowledgment receptions starting in state s_{n-1}, a state s is reached.
  If $s(\texttt{last\_unacknowledged}) \neq s_n(\texttt{next\_free})$, then the open window in state s is a subset of the open window in the state in which d is falsely acknowledged. Consequently, d is acknowledged in s. Hence, (b) is valid in state s_{n-1}. Contradiction!
  In case $s(\texttt{last\_unacknowledged}) = s_n(\texttt{next\_free})$, the transmit window in state s is again maximal. This again leads to a contradiction (either t_{n-1} is not enabled, or (c) is already valid in s_{n-1}).
- $t_{n-1} \in \{\texttt{receive\_ack}, \texttt{ack\_lost}\}$: In this case, every state which can be reached by reading acknowledgments was also reachable before from s_{n-1} by reading acknowledgments. Hence, (b) must also hold in s_{n-1}. This holds also for all subsequent cases, they are therefore left out.
- $t_{n-1} = \texttt{send\_ack}$: In this case, (d) holds in state s_{n-1}, since the new acknowledgment appended to the acknowledgment channel by R is the value of its variable `next_expected`.

Case: Property (c) holds in s_n. By reading the acknowledgments in the acknowledgment buffer, S can acknowledge erroneously, which means it can enlarge its transmit window. Again, consider transition t_{n-1}.

- $t_{n-1} \in \{\texttt{send}, \texttt{timeout}, \texttt{receive\_lost}, \texttt{receive}\}$: (c) already holds in state s_{n-1}. Contradiction!
- $t_{n-1} = \texttt{receive\_data}$: The sender is able to enlarge the open transmit window by reading some acknowledgments . Consider again the same reception sequence of acknowledgments but starting in s_{n-1} reaching a state s.
  If $s(\texttt{last\_unacknowledged}) = s_{n-1}(\texttt{next\_free})$, then in the sequence starting from s_n enlarging the window, the resulting open window size is 1. But this is also the size in s_n, thus, the open window is not enlarged.

Also if $s(\texttt{last\_unacknowledged}) = s_n(\texttt{next\_free})$, the open window size is not enlarged.

Otherwise, in both sequences, starting either from s_n or s_{n-1}, the open window size is enlarged by the same amount. Thus, (c) is already valid in s_{n-1}.

- $t_{n-1} = \texttt{send\_ack}$: In this case, (d) holds in state s_{n-1}.

Case: Property (d) holds in s_n. Assume that (a) does not hold. Then we can derive (using Lemma 3) that the output sequence is $0, 1, \ldots, i$ for a suitable $i \geq 0$. Consider transition t_{n-1}.

- $t_{n-1} \in \{\texttt{send}, \texttt{timeout}, \texttt{receive\_lost}\}$: (d) also holds in state s_{n-1}. Contradiction!
- $t_{n-1} = \texttt{send\_ack}$: Since the value of the variable `next_expected` is appended to the acknowledgment channel, (d) is also valid in s_{n-1}.
- $t_{n-1} = \texttt{receive\_data}$: Basically the same argumentation can be used as in the cases (b) and (c). One can think of an acknowledgment channel extended by the value of `next_expected`.
- $t_{n-1} = \texttt{receive}$: If the variable `next_expected` is unchanged by transition t_{n-1}, then (d) holds in s_{n-1}. Consequently, R must have received a message (d', n') with $n' = s_{n-1}(\texttt{next\_expected})$. Then, d' is given out by the receiver in this step, and there once was a state in which d' was in the transmit window on position n'.

  Assume first that (d) holds since the value of `next_expected` can be a bad acknowledgment in the sense of (b).

  Since (d) does not hold in s_{n-1}, the position in the transmit window in which the error occurs must be position n' (since t_{n-1} increases – modulo window size – the variable `next_expected` by 1). The output sequence (in state s_n) must be $0, 1, 2, \ldots, d'$.

  Hence, there is a data item $d'' > d'$ on position n'. Then, d' must have been acknowledged by the sender with a transition t_j with $j < n$. Since n is minimal, d' must already have been given out by R before step j. Consequently, d' is given out at least twice, once before transition t_j and once by taking transition t_{n-1}. Contradiction!

  Now assume that (d) holds since the value of `next_expected` would acknowledge erroneously, that means, enlarge the open transmit window. Thus, a state s is reachable by acknowledgment receptions such that an acknowledgment $n'' = s_n(\texttt{next\_expected})$ enlarges the open transmit window. Since $n' = \mathrm{pred}_m(n'')$, n' would also enlarge it in case $n' \neq s_n(\texttt{next\_free})$. But then, (d) would be valid in s_{n-1}. Consequently, $s_{n-1}(\texttt{next\_expected}) = n' = s_n(\texttt{next\_free}) = s_{n-1}(\texttt{next\_free})$. But then, position n' in the transmit window of s_{n-1} is not in the open transmit window. Thus, d' is already acknowledged. But it has not yet been given out in state s_{n-1}. Hence, (b) is valid in s_{n-1}.

4.6 Verification Results

The following table shows the experimental results on a SUN Ultra Sparc with 1 GB of memory and 167 MHz processor. The last column shows whether we used the Spin option for compression in this particular case. We always used partial order reduction.

Property	time	max. depth	memory	compression
(`Init`)	3 h	1,000,000	516 MB	no
(`WSize`)	15 h	1,000,000	901 MB	yes
(i)	1 min	1,675	80 MB	no
(ii)	1 h	2,781	139 MB	no
(iv)	1.5 h	1,000,000	259 MB	no

5 Conclusion

In this paper we proposed a verification methodology that combines data independence, abstraction, and compositional reasoning. The essence of the approach is to exploit the added value of combining those techniques. Additionally it serves as a clean guideline to separate the properties amenable to automatic model-checking and those to be verified deductively.

We applied this methodology to analyze a realistic sliding window protocol taken from a wireless ATM protocol of window size of 16. To our knowledge, this is the largest size of a sliding window protocol verified with the help of model-checking techniques. In fact we applied the same methodology to prove a liveness property of the same protocol.

Related Work Various versions of the sliding window protocol have been studied in the literature. The treatment ranges from informal arguments [Knu81] [SL92] et. al. (and already in the original proposal [Ste76] of the protocol), to model-checking (e.g. in [RRSV87]), compositional reasoning (e.g. [Jon87]), proofs using theorem provers [Car89], or combinations of various techniques [Kai97]. Closest to our investigation is [Kai97], who also uses a combination of model-checking, abstraction and decomposition. Different from our work, [Kai97] does not use a theorem prover, but automatically checks safety of the abstraction using specific behavioral preorders. He succeeds in proving safety and liveness up to the window size of 7.

Acknowledgments We thank the anonymous referees for their helpful comments and suggestions.

References

[BBLS92] A. Bouajjani, S. Bensalem, C. Loiseaux, and J. Sifakis. Property preserving simulations. In G. v. Bochmann and D. K. Probst, editors, *CAV'92*, volume 663 of *LNCS*. Springer, 1992.

[Car89] R. Cardell-Oliver. The specification and verification of sliding window protocols. Technical Report 183, University of Cambridge, 1989.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL, Los Angeles, CA*. ACM, 1977.

[CE81] E. M. Clarke and E. A. Emerson. Synthesis of synchronisation skeletons for branching time temporal logic. In D. Kozen, editor, *Workshop on Logic of Programs 1981*, volume 131 of *LNCS*. Springer, 1981.

[CGL94] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *TOPLAS*, 16(5):1512–1542, 1994.

[Dam96] D. Dams. *Abstract interpretation and partition refinement for model che cking*. PhD thesis, Technical University of Eindhoven, 1996.

[DGG94] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems: Abstractions preserving ACTL*, ECTL* and CTL*. In E.-R. Olderog, editor, *Proceedings of PROCOMET '94*. North-Holland, 1994.

[DPA+98] I. Dravapoulos, N. Pronios, A. Andristou, I. Piveropoulos, N. Passas, D. Skyrianoglou, G. Awater, J. Kruys, N. Nikaein, A. Enout, S. Decrauzat, T. Kaltenschnee, T. Schumann, J. Meierhofer, S. Thömel, and J. Mikkonen. *The Magic WAND, Deliverable 3D5, Wireless ATM MAC, Final Report*, 1998.

[dRLP98] W.-P. de Roever, H. Langmaack, and A. Pnueli, editors. *Compos '97*, volume 1536 of *LNCS*. Springer, 1998.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[Jon87] B. Jonsson. *Compositional Verification of Distributed Systems*. PhD thesis, University of Uppsala, 1987. Technical Report DoCS 87/09.

[Kai97] R. Kaivola. Using compositional preorders in the verification of sliding window protocol. In O. Grumberg, editor, *Proceedings of CAV '97*, volume 1256 of *LNCS*, pages 48–59. Springer, 1997.

[Kel95] P. Kelb. *Abstraktionstechniken für Automatische Verifikationsmethoden*. PhD thesis, University of Oldenburg, 1995.

[Knu81] D. E. Knuth. Verification of link-level protocols. *BIT*, 21:31–36, 1981.

[Kur94] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes, the automata theoretic approach*. Princeton Series in Computer Science. Princeton University Press, 1994.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1), 1995.

[Lon93] D. E. Long. *Model Checking, Abstraction, and Compositional Reasoning*. PhD thesis, Carnegie Mellon, 1993.

[ORSvH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[QS81] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium on Programming*, 1981.

[RRSV87] J. L. Richier, C. Rodriguez, J. Sifakis, and J. Voiron. Verification in Xesar of the sliding window protocol. In *PSTV VII*. North-Holland, 1987.

[SL92] A. U. Shankar and S. S. Lam. A stepwise refinement heuristics for protocol construction. *TOPLAS*, 14(3):417–461, 1992.

[Ste76] N. V. Stenning. A data transfer protocol. *Computer Networks*, 11:99–110, 1976.

[Wol86] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Thirteenth POPL (St. Peterburg Beach, FL)*, pages 184–193. ACM, 1986.