Compositional Verification of Timed Components using PVS[*]

Marcel Kyas
Christian-Albrechts-Universität zu Kiel, Germany
mky@informatik.uni-kiel.de

Jozef Hooman
Embedded Systems Institute & Radboud University Nijmegen, The Netherlands
hooman@cs.ru.nl

Abstract: We present a general framework to support the compositional verification of timed systems using the interactive theorem prover PVS. The framework is based on timed traces that are an abstraction of the timed semantics of flat UML state machines. We define a compositional proof rule for parallel composition and prove its soundness in PVS. After composition, a hiding rule can be applied to hide internal events. The general theories have been applied to parts of the Medium Altitude Reconnaissance System (MARS) as deployed in the F-16 aircraft of the Royal Netherlands Air-Force.

1 Introduction

In recent years, UML [Obj04] has been applied to the development of reactive safety-critical systems, in which the quality of the developed software is a key factor. Within the Omega project we have developed a method for the correct development of real-time embedded systems using a subset of UML, which consists of state machines, class diagrams, and object diagrams. In this paper we present a general framework supporting compositional verification of such designs using the interactive theorem prover PVS [ORS92, ORSvH95]. The framework is based on timed traces, which are abstractions of the timed semantics of UML state machines [vdZH06]. The focus is on the level of components and their interface specifications, without knowing their implementation [dR85, HdR85].

Our specifications are logical formulae that express the desired properties of a system or its components using predicates on timed traces. To formalise intermediate stages during the top-down design of a system, we have devised a mixed formalism where specifications and programming constructs can be mixed freely. In this paper, we restrict ourselves to parallel composition and hiding. This is inspired by similar work on untimed systems [Old85, Zwi89] and related to work on timed systems [Hoo98].

We apply our general theories to a part of the *Medium Altitude Reconnaissance System*

[*] This work has been supported by EU-project IST-2001-33522 OMEGA "Correct Development of Real-Time Embedded Systems." For more information, see `http://www-omega.imag.fr/`.

(MARS) as deployed by the Royal Netherlands Air Force on the F-16 aircraft [Ome05]. The system employs two cameras to capture high-resolution images. It counteracts image quality degradation caused by the aircraft's forward motion using a compensating motion of the film during its exposure. The control values for the forward motion compensation of the film speed and the frame rate are being computed in real-time, based on the current aircraft altitude, ground speed, and some additional parameters. The system is also responsible for producing the frame annotation, containing time and the aircraft's current position, which must be synchronised with the film motion. Here, we focus on the *databus manager*. It receives messages from sensors measuring the altitude and the position of the aircraft and tries to identify whether the sensors have broken down and — if they have — whether they have recovered.

In the OMEGA project, several formal techniques have been applied to the MARS case study. Live Sequence Charts (LSCs) [DH01] have been used to capture the requirements. Non-timed, functional properties of the MARS system have been verified using the model-checking tool UVE [STMW04]. Timed model checking has been applied by means of IFx, an extension of the IF toolbox [BGO$^+$04]. The approaches based on model-checking provide simulation and automated verification, but are limited to finite state systems.

To allow general verification of unbounded, infinite state systems, we have used the PVS tool, a general purpose theorem prover which is freely available [PVS]. PVS has a powerful specification language, based on higher-order typed logic. Specifications can be organised as hierarchies of parameterised theories, which may contain, e.g., declarations, definitions, axioms, and theorems. The PVS proof engine can be used to prove theorems which have been stated in the theories. To prove a particular goal, the user invokes proof commands which should simplify the goal until it can be proved automatically by PVS.

The first verification experiments with the original UML-model of the MARS system revealed that global, non-compositional verification is difficult and limited to small systems. To be able to apply compositional verification, the MARS system has been redesigned by means of a few well-defined components. The focus of this paper is on the specifications that have been used for the compositional verification of this redesign using PVS.

In the next section we describe the semantics of our formal framework. Section 3 introduces compositional proof rules. Section 4 describes the overall behaviour of our case study. Section 5 describes the decomposition of this overall specification into suitable components. Section 6 contains concluding remarks.

2 Semantics

Specifications are based on assertions which are predicates on traces θ consisting of observations o. For each observation we observe the *event* that is occurring, written $E(o)$, and the time at which it occurs, written $T(o)$. Time is defined to be a non-negative real and delays are assumed to be positive. The special event ϵ represents either that time elapses or that some hidden event is occurring. We use θ_i to denote the i-th observation of trace θ. Traces have to satisfy the following properties in order to be *well-formed*:

1. Time is monotone: $\forall i, j : i \leq j \to T(\theta_i) \leq T(\theta_j)$
2. Time progresses, i.e., is non-Zeno: $\forall i, \delta : \exists j : i \leq j \wedge T(\theta_i) + \delta \leq T(\theta_j)$
3. Proper events are instantaneous: $\forall i : E(\theta_i) \neq \epsilon \to T(\theta_i) = T(\theta_{i+1})$

The *projection* of a trace θ on a set of events $Eset$ is defined as:

$$\theta \downarrow Eset \overset{\text{def}}{=} \lambda k : \begin{cases} \theta_k, & \text{if } E(\theta_k) \in Eset \\ \epsilon, & \text{otherwise} \end{cases}$$

A *component* is specified by an assertion and a signature which is a set of events $Eset$ which can be observed by the component. Usually this concerns the receiving and the sending of messages. The assertion specifies the behaviour of the component, a set of traces, formalised by a predicate Θ on traces θ over its signature. Hence, a component C is defined by the pair $(Eset, \Theta)$, where the behaviour respects the interface, i.e., $\forall \theta : \Theta(\theta) \to \theta \downarrow Eset = \theta$.

We define *parallel composition* of components $C_1 = (E_1, \Theta_1)$ and $C_2 = (E_2, \Theta_2)$ as

$$C_1 \parallel C_2 \overset{\text{def}}{=} (E_1 \cup E_2, \{\theta \mid \theta \downarrow E_1 \in \Theta_1 \wedge \theta \downarrow E_2 \in \Theta_2 \wedge \theta \downarrow (E_1 \cup E_2) = \theta\})$$

That is, the projection of any trace of the parallel composition on the signature of one of the components yields a trace of this component. Observe that this implies that the components synchronise on their common events. Moreover, a trace of the composition should not include any new events outside the joint signature, as in [dRea01, Section 7.4].

For a component $C = (E, \Theta)$ and a set of events E' the *hiding operator* $C - E'$ removes the events in E' from the signature of C. It is formally defined by

$$C - E' \overset{\text{def}}{=} (E \setminus E', \{\theta \mid \exists \theta' \in \Theta : \theta = \theta' \downarrow (E \setminus E')\}).$$

We define a few suitable abbreviations.

- $E(\theta_i) = e$ states that the event e occurs at position i in the trace θ
- $\text{Never}(e, i, j)(\theta) \overset{\text{def}}{=} \forall k : i \leq k \wedge k \leq j \to E(\theta_k) \neq e$ asserts that the event e does not occur between positions i and j in the trace θ
- $\text{Never}(e)(\theta) \overset{\text{def}}{=} \forall k : E(\theta_k) \neq e$ asserts that e never occurs in trace θ
- $\text{AfterWithin}(e, i, \delta)(\theta) \overset{\text{def}}{=} \exists j : j \geq i \wedge E(\theta_j) = e \wedge T(\theta_j) - T(\theta_i) \leq \delta$ states that the event e occurs at some position j after i which is no later that δ time units from i

Because we aim at a mixed framework, in which specifications and programming constructs can be mixed freely, a *specification* is also considered to be a component. Hence specification $S = (E, \Theta)$ is identified with the component $(E, \{\theta \mid \theta \downarrow E = \theta \wedge \Theta(\theta)\})$.

Component $C_1 = (E_1, \Theta_1)$ *refines* component $C_2 = (E_2, \Theta_2)$, written $C_1 \implies C_2$, if $E_1 = E_2 \wedge \forall \theta : \Theta_1(\theta) \to \Theta_2(\theta)$. The refinement relation is a partial order on components and specifications.

3 Compositional Proof Rules

Next we derive a number of compositional proof rules. Their correctness is checked in PVS based on the semantic definitions and the definition of specifications.We start with a consequence rule, which allows the weakening of assertions in specifications.

Let $C_1 = (E_1, \Theta_1)$ and $C_2 = (E_2, \Theta_2)$ be two specifications. Then

$$(E_1 = E_2 \wedge (\forall \theta : \Theta_1(\theta) \rightarrow \Theta_2(\theta))) \rightarrow (C_1 \implies C_2)$$

To define a sound rule for parallel composition, we first show that the validity of an assertion Θ only depends on its signature. This is specified using the following predicate:

$$\text{depends}(\Theta, E) \overset{\text{def}}{\iff} \forall \theta, \theta' : \Theta(\theta) \wedge \theta \downarrow E = \theta' \downarrow E \rightarrow \Theta(\theta')$$

Then we can establish $\forall E : \text{depends}(\Theta, E) \leftrightarrow (\forall \theta : \Theta(\theta) \leftrightarrow \Theta(\theta \downarrow E))$. Using this statement we can prove the soundness of the following parallel composition rule:

$$\begin{aligned}&(\text{depends}(\Theta_1, E_1) \wedge \text{depends}(\Theta_2, E_2)) \rightarrow \\ &\quad ((E_1, \Theta_1) \parallel (E_2, \Theta_2) \implies (E_1 \cup E_2, \Theta_1 \wedge \Theta_2))\end{aligned}$$

To be able to use refinement in a context, we derive a monotonicity rule:

$$((C_1 \implies C_2) \wedge (C_3 \implies C_4)) \rightarrow ((C_1 \parallel C_3) \implies (C_2 \parallel C_4))$$

Similarly, we prove a compositional rule and a monotonicity rule for the hiding operator.

$$\text{depends}(\Theta, E_1 \setminus E_2) \rightarrow (((E_1, \Theta) - E_2) \implies (E_1 \setminus E_2, \Theta))$$

$$(C_1 \implies C_2) \rightarrow ((C_1 - E) \implies (C_2 - E))$$

4 The MARS Example

We consider only a small part of the MARS example, namely the *data bus manager*. This part serves as an illustration on how to apply the presented techniques to a timed system. Figure 1 shows the architecture of the data bus manager.

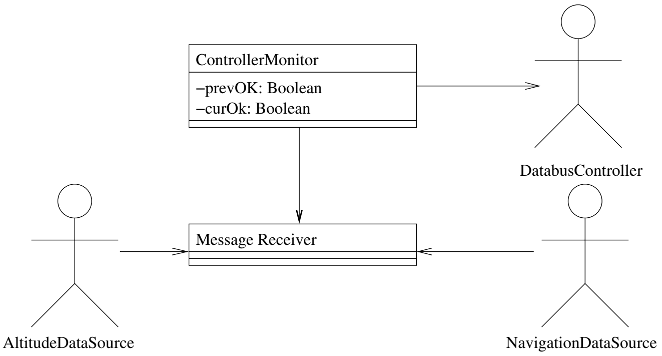

Figure 1: Architecture of data bus manager

The data sources *altitude data source* and *navigation data source* send data to a *message receiver*. If the data sources function correctly, they send data with period P and jitter $J < \frac{P}{2}$, as depicted in Figure 2; data should be sent in the grey periods.

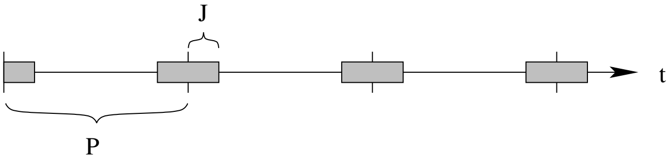

Figure 2: Data with period P and jitter J

First, we specify correct data sources, using $S = \{1, 2\}$ as an abstract representation of the two data sources. Let d_s represent the data items sent by source s, where s ranges over S, and $D = \{d_s \mid s \in S\}$ denotes the total set of data items sent by both sources.

For any data source s its behaviour is specified by the assertion $\mathrm{DS}_{s,1}(\theta) \wedge \mathrm{DS}_{s,2}(\theta)$ on its traces of observations θ. Assertion $\mathrm{DS}_{s,1}$ specifies that each occurrence of an event d_s is within the period specified by the jitter. $\mathrm{DS}_{s,2}$ specifies that at most one such message is sent during this period.

$$\mathrm{DS}_{s,1}(\theta) \overset{\text{def}}{\iff} \forall i : E(\theta_i) = d_s \rightarrow \exists n : nP - J \leq T(\theta_i) \wedge T(\theta_i) \leq nP + J$$

$$\begin{aligned}\mathrm{DS}_{s,2}(\theta) \overset{\text{def}}{\iff} & \ \forall i, j : E(\theta_i) = d_s \wedge E(\theta_j) = d_s \rightarrow \\ & \qquad i = j \vee P - 2J \leq |T(\theta_i) - T(\theta_j)|\end{aligned}$$

Consequently, a data source will not send data outside of the assigned time frame and will also not send more than one data sample during this time frame.

Next we formalise the global specification of the MARS system. If a data source fails to send a data item for K consecutive times, then the system shall indicate this error by sending signal err. The system is said to have recovered if N consecutive data messages have been received from each source. In the original MARS system $K = 3$ and $N = 2$.

The occurrence of N consecutive events e between i and j is specified by the predicate $\mathrm{occ}(e, N, i, j)$, which is defined as follows:

$$\begin{aligned}&\mathrm{occ}(e, N, i, j)(\theta) \overset{\text{def}}{\iff} N = 0 \vee \\ &\quad \exists f : |\,\mathrm{dom}(f)| = N \wedge f(0) = i \wedge \\ &\qquad f(|\,\mathrm{dom}(f)| - 1) = j \wedge (\forall k : k \leq |\,\mathrm{dom}(f)| - 1 \rightarrow E(\theta_{f(k)}) = e) \wedge \\ &\qquad (\forall k : k < |\,\mathrm{dom}(f)| - 1 \rightarrow f(k) < f(k+1) \wedge \\ &\qquad P - J < T(\theta_{f(k+1)}) - T(\theta_{f(k)}) \wedge T(\theta_{f(k+1)}) - T(\theta_{f(k)}) < P + J)\end{aligned}$$

This implies that there exists a strictly monotonically increasing sequence f of length N of indexes starting at i and ending at j such that at each position in this sequence the event e occurs and that these events occur $P \pm J$ time-units apart.

To express that K data items have been missed we define:

$$\mathrm{TimeOut}(e, t, i, j)(\theta) \overset{\text{def}}{\iff} \mathrm{Never}(e, i, j) \wedge T(\theta_j) - T(\theta_i) \geq t$$

which states that event e has not occurred for at least t time units between positions i and j in trace θ.

Observe that a data source s is in an error state at position i in the trace θ if it has not sent data for at least $L \overset{\text{def}}{=} KP + 2J$ time units at position $j \leq i$ and that it has not recovered until position i. This is expressed by the following assertion:

$$\begin{aligned}\mathrm{Error}(d,i)(\theta) \overset{\text{def}}{\iff}\ & \exists k,j : j \leq i \wedge \mathrm{TimeOut}(d,L,k,j)(\theta) \wedge \\ & \quad (\forall m : j < m \wedge m \leq i \rightarrow \neg\exists l : \mathrm{occ}(d,N,l,m)(\theta))\end{aligned}$$

The validity of an error signal is specified by assertion TDS_1, where Δ_{err} represents the delay needed to react to the occurrence of an error.

$$\begin{aligned}\mathrm{TDS}_1(\theta) \overset{\text{def}}{\iff}\ & \forall i,j : i < j \wedge (\exists s : \mathrm{TimeOut}(d_s,L,i,j)(\theta)) \wedge \\ & \quad (\forall s : \neg\,\mathrm{Error}(d_s,j)(\theta)) \rightarrow \mathrm{AfterWithin}(\mathit{err},j,\Delta_{\mathit{err}})(\theta)\end{aligned}$$

The integrity of the error signal *err* is specified by:

$$\begin{aligned}\mathrm{TDS}_2(\theta) \overset{\text{def}}{\iff}\ & \forall j : E(\theta_j) = \mathit{err} \rightarrow \\ & \quad \exists i,k : i < k \wedge k < j \wedge (\exists s : \mathrm{TimeOut}(d_s,L,i,k)(\theta)) \wedge \\ & \qquad (\forall s : \neg\,\mathrm{Error}(d_s,k)(\theta)) \wedge \mathrm{Never}(\mathit{err},k,j-1)(\theta)\end{aligned}$$

The system recovers from an error when all data sources have been sending N consecutive messages. This recovery is indicated by sending a ok signal. The next predicate specifies that all sources have indeed sent N consecutive data messages.

$$\begin{aligned}\mathrm{Recover}(D,i,j) \overset{\text{def}}{\iff}\ & \exists f,g : i = \min_{d\in D} f(d) \wedge j = \max_{d\in D} g(d) \wedge \\ & \quad (\forall d,d' : |T(\theta_{f(d)}) - T(\theta_{f(d')})| \leq 2J) \wedge \\ & \quad (\forall d,d' : |T(\theta_{g(d)}) - T(\theta_{g(d')})| \leq 2J) \wedge \\ & \quad (\forall d : \mathrm{occ}(d,N,f(d),g(d)))\end{aligned}$$

This predicate states that there exist two functions f and g from events to positions such that i is the smallest value produced by f, j is the largest value produced by g, the values in the range of f are at most $2J$ time units apart, as are the values in the range of g such that we have N occurrences of d between $f(d)$ and $g(d)$. Using this predicate, we can define the validity of the ok signal, using delay Δ_{ok} to model the reaction time needed to recover.

$$\begin{aligned}\mathrm{TDS}_3(\theta) \overset{\text{def}}{\iff}\ & \forall i,j : i < j \wedge \mathrm{Recover}(D,i,j)(\theta) \wedge \\ & \quad (\exists s : \mathrm{Error}(d_s,j)(\theta)) \rightarrow \mathrm{AfterWithin}(ok,j,\Delta_{ok})(\theta)\end{aligned}$$

The integrity of the ok signal is specified by:

$$\begin{aligned}\mathrm{TDS}_4(\theta) \overset{\text{def}}{\iff}\ & \forall j : E(\theta_j) = ok \rightarrow \\ & \quad \exists i,k : i < k \wedge k < j \wedge (\exists s : \mathrm{Error}(d_s,i)(\theta)) \wedge \\ & \qquad \mathrm{Recover}(D,i,k)(\theta) \wedge \mathrm{Never}(ok,k,j-1)(\theta)\end{aligned}$$

Finally, we specify the behaviour of the global system by TDS:

$$\mathrm{TDS}(\theta) \overset{\text{def}}{\iff} \bigwedge_{1\leq i\leq 4} \mathrm{TDS}_i(\theta)$$

5 Decomposition of the MARS example

In this section we decompose the MARS system in a few components, such that we can show by compositional deductive verification that the composition of these components satisfies the global specification TDS as presented in the previous section.

The main idea is that we specify a separate data receiver for each data source s and later compose these receivers for different data sources with a component that specifies the combinations of errors and recovery. This architecture is depicted in Figure 3.

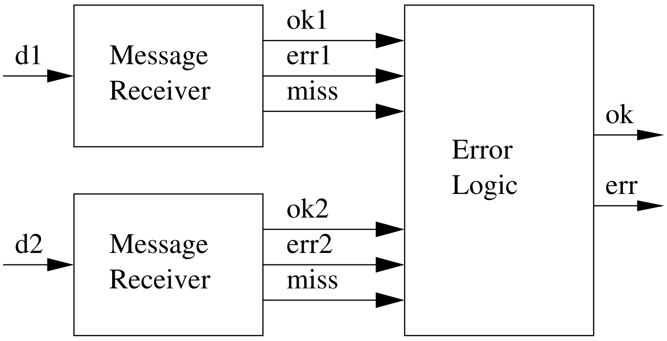

Figure 3: Decomposed architecture for two data sources

The *message receivers* are identical processes; for a data source s it receives data items d_s and internal states are made visible by external signals err_s, $miss$, and ok_s to represent error and recovery. The role of $miss$ signals will be explained later.

5.1 Message Receiver

The message receiver processes the data received from one data source. Processing data takes time, which varies depending on the data received. We assume that this time is between l and u. The message receiver should enter an error state if K successive messages are missing from its source. It should resume normal operation if it has received N successive messages from its source. Observe that this is very similar to the global specification of the MARS system, now restricted to a single data source. Hence the assertions $\mathrm{MR}_{s,1}$ through $\mathrm{MR}_{s,4}$ which specify the err_s and ok_s events are similar to TDS_1 through TDS_4. Here we only present $\mathrm{MR}_{s,5}$ and $\mathrm{MR}_{s,6}$ which specify the $miss$ event.

The error logic component, to be specified in the next subsection, has to be notified by a message receiver that did not receive a data message in time. This is indicated by a $miss$ message, which has to be introduced because using only err and ok signals is not sufficient for recovery according to the specification. The problem is that the err signal indicates the absence of K data items, whereas recovery requires the presence of N consecutive data signals from the data source. Observe that, when staying in the correct operational mode, a few missing data items are allowed, but no missing data item is allowed when trying to recover.

Observe that we can use a single $miss$ event for all message receivers. We do not need a separate event for each message receiver, because in order to recover, *all* message receivers have to receive N consecutive data messages. The $miss$ signal indicates that there exists a component which missed a data message during this period. The error logic component need not know which message receiver missed the data message.

A message receiver sends a $miss$ message to the error logic whenever a time-out for a data message occurs *and* it is not in an error state. If the message receiver is already in an error state, it signals N consecutive data messages using an ok message. Therefore, it is not necessary to send $miss$ signals in this case. Sending a $miss$ signal may be delayed by at most $\Delta^{\mathrm{MR}}_{miss}$ time units.

$$\begin{aligned}\mathrm{MR}_{s,5}(\theta) \overset{\text{def}}{\iff} \forall j : & \mathrm{TimeOut}(d_s, P + 2J, i, j)(\theta) \wedge \neg\,\mathrm{Error}(d_s, j)(\theta) \rightarrow \\ & \mathrm{AfterWithin}(miss, j, \Delta^{\mathrm{MR}}_{miss})(\theta)\end{aligned}$$

Note that if the Kth data item is missed at j, the $\mathrm{Error}(d_s, j)(\theta)$ predicate is true and signal $miss$ is not emitted. Instead, by $\mathrm{MR}_{s,3}$, an err_s signal is sent, i.e., not both a $miss$ signal and an err_s signal are sent.

The integrity of a $miss$ event is specified by $\mathrm{MR}_{s,6}$.

$$\begin{aligned}\mathrm{MR}_{s,6}(\theta) \overset{\text{def}}{\iff} \forall j : E(\theta_j) = miss \rightarrow & \\ \exists i, k : i < k \wedge k < j \wedge \neg\,\mathrm{Error}(d_s, i)(\theta) \wedge & \\ \mathrm{TimeOut}(d_s, P + 2J, i, k)(\theta) \wedge \mathrm{Never}(miss, k, j-1)(\theta) &\end{aligned}$$

From N missing $miss$ signals one can conclude that the data source s has received N consecutive data messages:

Lemma 1. For any s, i, j, if $\mathrm{TimeOut}(miss, NP + 2J, i, j)$ then $\mathrm{occ}(d_s, N, i, j)$

More importantly, the timeout of the $miss$ signal implies that all message receivers have received N consecutive data messages.

Corollary 2. For all i, j, if $\mathrm{TimeOut}(miss, NP + 2J, i, j)$, then $\mathrm{Recover}(D, i, j)$

Finally, we specify a message receiver for a source s as $\mathrm{MR}_s(\theta) \overset{\text{def}}{\iff} \bigwedge_{1 \le i \le 6} \mathrm{MR}_{s,i}(\theta)$.

5.2 Error Logic

The error logic component accepts err_s and ok_s signals from each data source s, as well as a signal $miss$ indicating that there exists a data source s that has not received data from its source during this cycle. The error logic will emit an err signal if it detects an error in the system and an ok signal if the system recovers after an error. The behaviour of the error logic is specified in the state machine of Figure 4.

State AllOk indicates that the system operates normally. When receiving an err_s signal

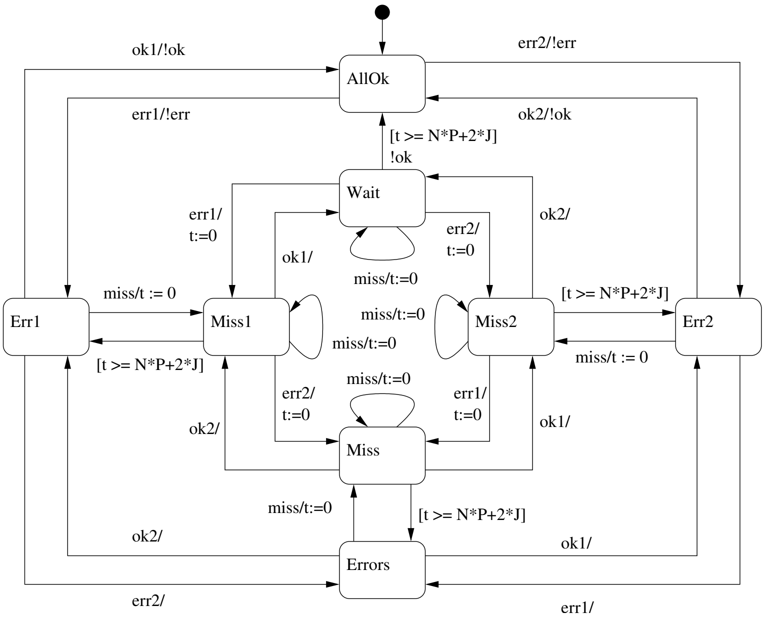

Figure 4: State Machine of the error logic component

in this state, for some $s \in \{1, 2\}$, signal err is sent and state Err_s is entered. Recovery from this state occurs when an ok_s signal is received. But if an error signal is received for the other source, state Errors is entered, indicating that the system has to recover from an error in both data sources.

As long as the system is in the AllOk state it ignores all $miss$ signals. If the system is in an error state, i.e., one of err_1, err_2, or Errors, and it receives a $miss$ signal, the error logic has to wait for a *new* time-out of the miss signal and the required number of ok signals in order to return to normal operation, which is represented by the states Miss_1, Miss_2, Miss, and Wait. The system measures the time elapsed since the latest reception of a $miss$ signal using the clock t. Consequently, it resets t whenever it receives a $miss$ signal *or* a err_s signal.

State Wait is entered whenever one of the Miss_s states has been left after receiving the corresponding ok_s signal, and the error logic itself has to emit an ok signal to confirm that the system has recovered from the error condition. Observe that we have to wait until the end of the current period in order to assert that during this time neither message receiver sends an error signal. After a time-out of the $miss$ signal, state Wait is left, AllOk is entered, and an ok signal is emitted.

As an example, we give a scenario to show that state Wait is reachable. Suppose an err_1 signal is received in state AllOk, leading to state Err_1. During the next period a $miss$ signal is received from message receiver 2. This causes a state change to Miss_1, indicating that it has to receive an ok_1 signal and, moreover, has to wait until message receiver 2 received N consecutive data messages. Observe that in this situation message receiver

1 only has to receive $N - 1$ data messages. Assuming that both message receivers will receive their data messages, message receiver 1 sends its ok_1 signal after $N - 1$ periods, after which state Wait is entered. Next the error logic component has to wait another period in order to make sure that message receiver 2 has received its Nth data message, after which it may signal recovery.

In order to specify the error logic component in a declarative way, we first formalise whether an error of data source s has been detected and when the system is in state AllOk.

$$\mathrm{Error}(i, s)(\theta) \overset{\text{def}}{\Longleftrightarrow} \exists m : m \leq i \wedge E(\theta_m) = err_s \wedge \mathrm{Never}(ok_s, m+1, i)(\theta)$$

$$\mathrm{AllOk}(i)(\theta) \overset{\text{def}}{\Longleftrightarrow} \forall s : \neg\, \mathrm{Error}(i, s)(\theta)$$

The validity and integrity of an err signal indicating error is specified as follows, using a maximal delay of $\Delta^{\mathrm{EL}}_{err}$ time units.

$$\begin{aligned}\mathrm{EL}_1(\theta) \overset{\text{def}}{\Longleftrightarrow} \forall i : & \mathrm{AllOk}(i)(\theta) \wedge (\exists s : E(\theta_{i+1}) = err_s) \rightarrow \\ & \mathrm{AfterWithin}(err, i+1, \Delta^{\mathrm{EL}}_{err})\end{aligned}$$

$$\begin{aligned}\mathrm{EL}_2(\theta) \overset{\text{def}}{\Longleftrightarrow} \forall j : & E(\theta_j) = err \rightarrow \\ & \exists i : i < j \wedge \mathrm{AllOk}(i)(\theta) \wedge (\exists s : E(\theta_{i+1}) = err_s) \wedge \\ & \quad \mathrm{Never}(err, i+2, j-1)(\theta)\end{aligned}$$

The next predicate states that a data source s recovers from an error:

$$\mathrm{Recover}(i, s)(\theta) \overset{\text{def}}{\Longleftrightarrow} \forall i : \mathrm{Error}(i-1, s)(\theta) \wedge E(\theta_i) = ok_s$$

Next, using a maximal delay of $\Delta^{\mathrm{EL}}_{ok}$, the validity and integrity of an ok signal is specified.

$$\begin{aligned}\mathrm{EL}_3(\theta) \overset{\text{def}}{\Longleftrightarrow} \forall i : & (\exists s : \mathrm{Recover}(i, s)(\theta)) \wedge (\forall s : \neg\, \mathrm{Error}(i, s)) \wedge \\ & (\exists k : \mathrm{TimeOut}(miss, NP{+}2J, k, i)(\theta)) \rightarrow \mathrm{AfterWithin}(ok, i, \Delta^{\mathrm{EL}}_{ok})\end{aligned}$$

$$\begin{aligned}\mathrm{EL}_4(\theta) \overset{\text{def}}{\Longleftrightarrow} \forall j : & E(\theta_i) = ok \rightarrow \\ & \exists i : i < j \wedge (\exists s : \mathrm{Recover}(i, s)(\theta)) \wedge (\forall s : \neg\, \mathrm{Error}(i, s)) \wedge \\ & \quad \mathrm{Never}(ok, i{+}1, j{-}1) \wedge (\exists k : \mathrm{TimeOut}(miss, NP{+}2J, k, i)(\theta))\end{aligned}$$

The error logic is specified by the assertion: $\mathrm{EL}(\theta) \overset{\text{def}}{\Longleftrightarrow} \bigwedge_{1 \leq i \leq 4} \mathrm{EL}_i(\theta)$

6 Conclusions

We have presented a compositional framework for the compositional verification of high-level real-time components which communicate by means of events. Compositional proof rules for parallel composition and hiding have been proved sound in PVS. In this way, we can use deductive verification in PVS to prove the correctness of a decomposition of a system into a number of communicating components. Next, the components can be implemented independently using UML, according to their specification, and the correctness of the implementation with respect to the interface specification may be established by means of other techniques, such as model checking.

The framework has been applied to the MARS case study, which has been supplied by the Netherlands National Aerospace Laboratory in the form of UML models. The specifications presented here are the result of a long and arduous path leading to consistent specifications of the parts and the full formal proof in PVS. In general, interactive verification of UML models is very complex because we have to deal with many features simultaneously, such as timing, synchronous operation calls, asynchronous signals, threads of control, and hierarchical state machines. Hence, compositionality and abstraction are essential to improve scalability. Verifying the MARS case study indeed shows that deductive verification is more suitable for the correctness proofs of high-level decompositions, to eventually obtain relatively small components that are suitable for model checking.

Since the original UML model of MARS was monolithic, a redesign of the original system was necessary to enable the application of compositional techniques and increase our understanding of the model. Interestingly, this led to a design that is more flexible, e.g., for changing the error logic, and more easily extensible, e.g., to more data sources, than the original model.

Errors in the decomposition of the MARS system have been found using model checking (by means of the IF validation environment [BFG$^+$00] and UPPAAL [LPY95]) and by the fact that no proof could be found for the original specification. One of these errors was that we did not include a $miss$ signal, which is required to correctly observe recovery in the error logic component. Otherwise, the system recovered in circumstances where the global specification did not allow this.

Observe that the compositional approach requires substantial additional effort to obtain appropriate specifications for the components. Finding suitable specifications is difficult. Hence, it is advisable to start with finite high-level components and to simulate and to model-check these as much as possible. Apply interactive verification only when sufficient confidence has been obtained. Finally, it is good to realise that interactive verification is quite time consuming and requires detailed knowledge of the tool.

Acknowledgements We would like to thank all partners of the OMEGA project for many fruitful discussions on the MARS case study.

References

[BFG$^+$00] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, and Laurent Mounier. IF: A Validation Environment for Timed Asynchronous Systems. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification '00*, volume 1855 of *LNCS*. Springer-Verlag, 2000.

[BGO$^+$04] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. The IF toolset. In *SFM-04:RT 4th Int. School on Formal Methods for the Design of Computer, Communication and Software Systems: Real Time*, pages 237–267. LNCS 3185, Springer-Verlag, 2004.

[dBdRR85] Jaco W. de Bakker, Willem-Paul de Roever, and Grzegorz Rozenberg, editors. *Current Trends in Concurrency*, volume 224 of *LNCS*. Springer-Verlag, 1985.

[DH01] Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[dR85] Willem-Paul de Roever. The Quest for Compositionality — a survey of assertion-based proof systems for concurrent programs, Part 1: Concurrency based on shared variables. In *Proc. IFIP Working Conference 1985: The Role of Abstract Models in Computer Science*. North-Holland, 1985.

[dRea01] Willem-Paul de Roever et al. *Concurrency Verification*. Cambridge University Press, 2001.

[HdR85] Jozef Hooman and Willem-Paul de Roever. The Quest goes on: A survey of Proof Systems for Partial Correctness of CSP. In de Bakker et al. [dBdRR85].

[Hoo98] Jozef Hooman. Compositional Verification of Real-Time Applications. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference*, volume 1536 of *LNCS*. Springer-Verlag, 1998.

[LPY95] Kim Larsen, Paul Pettersson, and Wang Yi. Model-Checking for Real-Time Systems. In Horst Reichel, editor, *Proc. Fundamentals of Computation Theory*, volume 965 of *LNCS*. Springer-Verlag, 1995.

[Obj04] Object Management Group. *UML 2.0 Superstructure Specification*, October 2004. `http://www.omg.org/cgi-bin/doc?ptc/2004-10-02`.

[Old85] Ernst-Rüdiger Olderog. Process Theory: Semantics, specifications and verification. In de Bakker et al. [dBdRR85].

[Ome05] Omega Consortium. Medium Altitude Reconnaissance System. Webpage at `http://www-omega.imag.fr/cs/MARS/MARS.php`, 2005.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *Automated Deduction – CADE-11*, volume 607 of *LNAI*. Springer-Verlag, 1992.

[ORSvH95] Sam Owre, John M. Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software*, 21(2):107–125, 1995.

[PVS] PVS. `http://pvs.csl.sri.com/`.

[STMW04] Ingo Schinz, Tobe Toben, Christian Mrugalla, and Bernd Westphal. The Rhapsody UML Verification Environment. In *Proc. 2nd IEEE Int. Conf. on Software Engineering and Formal Methods (SEFM2004)*, pages 174–183. IEEE Computer Society Press, 2004.

[vdZH06] Mark van der Zwaag and Jozef Hooman. A Semantics of Communicating Reactive Objects with Timing. *Journal on Software Tools for Technology Transfer*, 2006.

[Zwi89] Job Zwiers. *Compositionality, Concurrency and Partial Correctness*, volume 321 of *LNCS*. Springer-Verlag, 1989.