# Verifying OCL specifications of UML models : tool support and compositionality

Kyas, M.

# Chapter 7

# Compositional Verification of Timed Components in PVS

This chapter contains the compositional modelling and verification of the error logic part of the MARS case study [117] in PVS, based on Omega Deliverable 3.3 Appendix 42. We also extend the theory presented in Chapter 6 with real-time.

Essentially, the theory used in this chapter is based on the theory presented in [65]. This theory has been adapted to the needs of object-oriented programming, as presented in the preceding chapters, to deal with timed systems.

One crucial observation is that in the presence of time every event has to be observed when it occurs, and not, like we have done with object creation in Chapter 6, later. The reason is that we also observe the time when some event occurs.

## 7.1 Introduction

In recent years, UML [111] 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 [116] 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 to support compositional verification of such designs defined in this subset of UML using the interactive theorem prover PVS [121, 122]. The framework is based on timed traces. These are an abstraction of the timed semantics of UML state machines as described in [149]. The focus is on the level of components and their interface specifications, without knowing their implementation [44, 66].

Our specifications are based on assertions on timed traces, that is, logical formulae that express the desired properties of a system or one of its components. To be able to formalise intermediate stages during the top-down design of a system we aim at a mixed formalism, that is, a 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 [114, 115, 158] and related to work on timed systems [65, 139].

We apply these general theories to a case study, namely, part of the *Medium Altitude Reconnaissance System* (MARS) deployed by the Royal Netherlands Air Force on the F-16 aircraft [117]. The system employs two cameras to capture high-resolution images. It counteracts image quality degradation caused by the forward motion of an aircraft by creating a compensating motion of the film during the film exposure. The controls applied to the camera for the film speed of the *forward motion compensation* 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. Finally, the system performs health monitoring and alarm processing functions. The part of this case study we focus on the *data-bus manager*. It receives messages from sensors measuring the altitude and its position and tries to identify whether the sensors have broken down, and if they have, whether they have recovered.

This paper is structured as follows. In the next section we describe the semantics of our assertion language. Section 7.3 defines our proof rules. Section 7.4 describes the overall behaviour of our case study. Section 7.5 describes the decomposition of this overall specification into suitable components. Section 7.6 gives a high-level view how the correctness of the decomposition is proved. For the actual proof see [87]. The final Section 7.7 contains some concluding remarks, especially regarding the arduous path leading up to our presented results.

## 7.2 Semantics

Assertions are predicates on traces $\theta$ consisting of observations $o$. For each such observation we observe the *event* that is occurring, written as $E(o)$, and the time at which it occurs, written as $T(o)$. Time is defined to be a non-negative real, and delays are assumed to be positive. The special event $\epsilon$ represents either that time elapses or that some hidden event is occurring.

These traces have to satisfy the following properties in order to be *well-formed*:

1. Time is monotone: $\forall i, j : i \leq j \implies T(\theta_i) \leq T(\theta_j)$.
2. Time progresses, that is, 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: $\forall i : E(\theta_i) \neq \epsilon \implies T(\theta_i) = T(\theta_{i+1})$.

Next, we define the projection of a trace $\theta$ on a set of events $E$:

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

Observe that this projection operator is idempotent, that is, $(\theta \downarrow E) \downarrow E = \theta \downarrow E$ and satisfies: $\forall \theta, E_1, E_2 : \theta \downarrow E_1 = \theta \wedge E_1 \subseteq E_2 \implies \theta \downarrow E_2 = \theta$.

Next we define the notion of a component. A component specifies a set of events $E$ as its signature, that is, as a set of events which a component is able to observe and react to. Usually, these events will be the receiving and sending of messages. As its behaviour the component specifies a set of traces, formalised by a predicate $\Theta$ on traces $\theta$ over its signature. A component $C$ is defined to be the pair $(E, \Theta)$. For any component $C = (E, \Theta)$, we require that its behaviour respects its interface: $\forall \theta : \Theta(\theta) \implies \theta \downarrow E = \theta$.

We define the parallel composition of components $C_1 = (E_1, \Theta_1)$ and $C_2 = (E_2, \Theta_2)$ as $C_1 \parallel C_2 \stackrel{\text{def}}{=} (E_1 \cup E_2, \{\theta \mid \theta \downarrow E_1 \in \Theta_1 \wedge \theta \downarrow E_2 \in \Theta_2 \wedge \theta \downarrow (E_1 \cup E_2) = \theta\})$. That is, the parallel composition of two components maintains the behaviour of its parts, the components synchronise on their common events, and it does not include new events outside the common signature, as in [45, 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: $C - E' \stackrel{\text{def}}{=} (E \setminus E', \{\theta \mid \exists \theta' \in \Theta : \theta = \theta' \downarrow (E \setminus E')\})$.

The behaviour of a component is specified by *assertions*, which are predicates on traces. We lift the boolean connectives to assertions in the usual manner.

We define a few suitable abbreviations. The term $E(\theta_i) = e$ states that the event $e$ occurs at position $i$ in the trace $\theta$. The term

$$\text{Never}(e, i, j)(\theta) \stackrel{\text{def}}{=} \forall k : i \leq k \wedge k \leq j \implies E(\theta_k) \neq e$$

asserts that the event $e$ does not occur between positions $i$ and $j$ in the trace $\theta$. Similarly, the assertion

$$\text{Never}(e)(\theta) \stackrel{\text{def}}{=} \forall k : E(\theta_k) \neq e$$

asserts, that $e$ never occurs in a trace. Finally,

$$\text{AfterWithin}(e, i, \delta)(\theta) \stackrel{\text{def}}{=} \exists j : j \geq i \wedge E(\theta_j) = e \wedge T(\theta_j) - T(\theta_i) \leq \delta$$

is an assertion that states that the event $e$ occurs at some position $j$ after $i$ which is no later that $\delta$ time units from $i$.

*Specifications* of components consist of a signature and an assertion. 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. Therefore a specification $S = (E, \Theta)$ is identified by the component $(E, \{\theta \mid \theta \downarrow E = \theta \wedge \Theta(\theta)\})$, which has the same name.

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

We have chosen to use the implication symbol $\implies$ for refinement, because in out theory, refinement (almost) *is* implication. The crucial part of our definition of refinement of components is $\Theta_1(\theta) \implies \Theta_2(\theta)$! The same idea also holds for Lamport's TLA [89].

## 7.3 Compositional Proof Rules

In this section we derive a number of compositional proof rules. Their correctness is checked in PVS based on the semantic definitions and the definition of specifications [87]. 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

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

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

$$\text{depends}(\Theta, E) \stackrel{\text{def}}{\iff} \forall \theta, \theta' : \Theta(\theta) \wedge \theta \downarrow E = \theta' \downarrow E \implies \Theta(\theta') \quad .$$

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

$$(\text{depends}(\Theta_1, E_1) \wedge \text{depends}(\Theta_2, E_2)) \implies ((E_1, \Theta_1) \parallel (E_2, \Theta_2) \implies (E_1 \cup E_2, \Theta_1 \wedge \Theta_2)) \quad .$$

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

$$((C_1 \implies C_2) \wedge (C_3 \implies C_4)) \implies ((C_1 \parallel C_3) \implies (C_2 \parallel C_4)) \quad .$$

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

$$\text{depends}(\Theta, E_1 \setminus E_2) \implies (((E_1, \Theta) - E_2) \implies (E_1 \setminus E_2, \Theta))$$
$$(C_1 \implies C_2) \implies ((C_1 - E) \implies (C_2 - E)) \quad .$$

## 7.4 The MARS Example

From the MARS example we consider only a small part, namely the *data bus manager*. This part serves as an illustration on how to apply the presented techniques to a timed system. Figure 7.1 shows the architecture of the data bus manager.

The external data sources *altitude data source* and a *navigation data source* send data, here represented by abstract events $d_1$ and $d_2$, respectively, to a *message receiver*. If the sources function correctly, they send data with period $P$ and jitter $J < \frac{1}{2}P$, as depicted in Figure 7.2: Data should be available during the grey periods.

For any data source $s$ its behaviour can be specified by the assertion $\text{DS}_{s,1}(\theta) \wedge \text{DS}_{s,2}(\theta)$ on its traces of observations $\theta$, where $\text{DS}_{s,1}$ and $\text{DS}_{s,2}$ are defined below. The

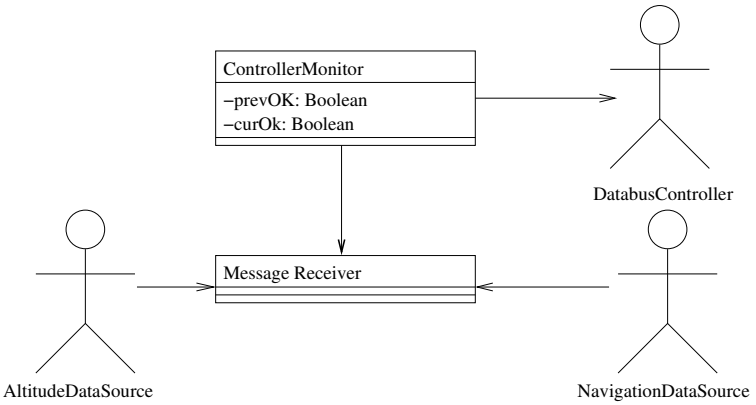


**Figure 7.1:** Architecture of the data bus manager

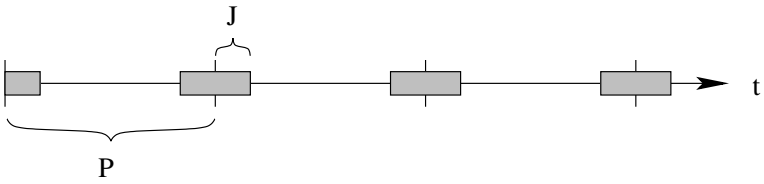


**Figure 7.2:** Data with period $P$ and jitter $J$

assertion $\mathrm{DS}_{s,1}$, where $s$ ranges over $1, 2$, specifies that each occurrence of an event $d_s$ is within the period specified by the jitter. The assertion $\mathrm{DS}_{s,2}$ specifies that at most one such message is sent during this period:

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

$$\mathrm{DS}_{s,2}(\theta) \overset{\text{def}}{\iff} \forall i, j : E(\theta_i) = d_s \wedge E(\theta_j) = d_s \implies i = j \vee P - 2J \leq |T(\theta_i) - T(\theta_j)| \quad .$$

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. A state machine depicting such normal behaviour is shown in Figure 7.3. The data source may send data only while it is in the state Initial or in the state Send. While it is in the Initial state, it may stay in this state for at most $J$ time units.

If a data source fails to send a data item for $K$ consecutive times, then the bus manager shall indicate the error by sending an *err* signal. That this situation has occurred is formalised by an appropriate timeout assertion:

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

This assertion states that the event $e$ has not occurred for at least $t$ time units between positions $i$ and $j$ of a trace $\theta$.

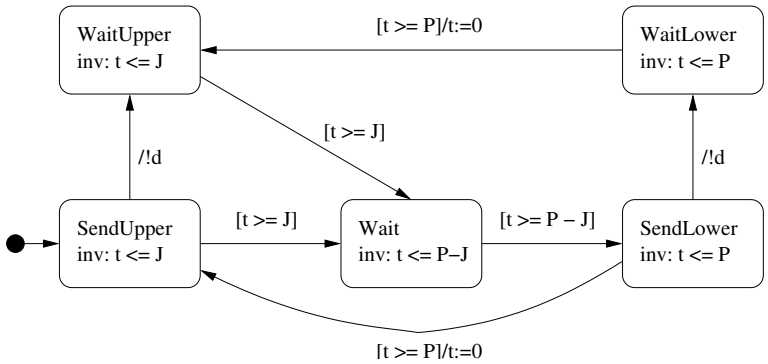


**Figure 7.3:** State machine of the data source

The system is said to have recovered, if $N$ consecutive data messages have been received from each source. 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:

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

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.

We can now define that a data source $s$ is in an error state at position $i$ in the trace $\theta$ by observing that 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 $i$:

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

The validity of an error signal is specified by the following predicates:

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

Here, $\Delta_{err}$ expresses a delay that models the time the system needs to react to the

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

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

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 from $D = \{d_s \mid s \in S\}$:

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

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:

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

Similar to $\Delta_{err}$, the delay $\Delta_{ok}$ models the time needed by the error logic to react to the recovery of the system. The integrity of the *ok* signal is specified by:

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

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

$$\mathrm{TDS}(\theta) \overset{\text{def}}{\iff} \mathrm{TDS}_1(\theta) \wedge \mathrm{TDS}_2(\theta) \wedge \mathrm{TDS}_3(\theta) \wedge \mathrm{TDS}_4(\theta) \quad .$$

From this global specification we derive a version of the data bus manager in such a way that full compositional deductive verification becomes possible and the verification task is split into smaller verification tasks.

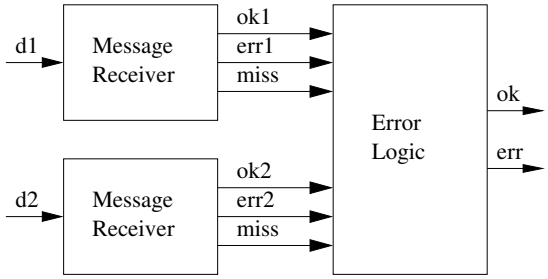


**Figure 7.4:** Decomposed architecture for two data sources

## 7.5 Decomposition of the MARS example

The main idea is that we specify a separate data receiver for each data type $d$ and later compose the receivers for different data sources with a component that specifies the combinations of errors and recovery. This architecture is depicted in Figure 7.4.

The *message receivers* are two identical processes, whose internal states are made visible by external signals *err*, *miss*, and *ok* to represent error and recovery. Hence the message receiver is specified by a component, parameterised over events $d$, *err*, *miss*, and *ok*. The role of *miss* signals will be explained later.

### 7.5.1 Message Receiver

The message receiver processes the data received from one data source. Processing data takes some 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 if $K$, say 3, successive messages are missing from its source. It should resume normal operation if it has received $N$, say 2, successive messages from its source. This behaviour is depicted by the state machine in Figure 7.5.

The message receiver receives data from one data source and counts how many consecutive messages have been absent from the source. We can assert that an error is present once no message has been received since $L$ time units. Recall, that this is the time required to observe that $K$ messages have been absent from the input stream. If this occurs, the message receiver sends a $err_s$ message to an error logic component to indicate to it that from one source $K$ messages have been missing. The delay $\Delta^{\mathrm{MR}}_{err}$ models the time needed by the message receiver to send its $err_s$ signal. This is specified by:

$$\mathrm{MR}_{s,1}(\theta) \stackrel{\mathrm{def}}{\Longleftrightarrow} \forall i, j : \mathrm{TimeOut}(d_s, L, i, j)(\theta) \wedge \neg\, \mathrm{Error}(d_s, i)(\theta) \implies \mathrm{AfterWithin}(err_s, j, \Delta^{\mathrm{MR}}_{err})(\theta) \quad .$$

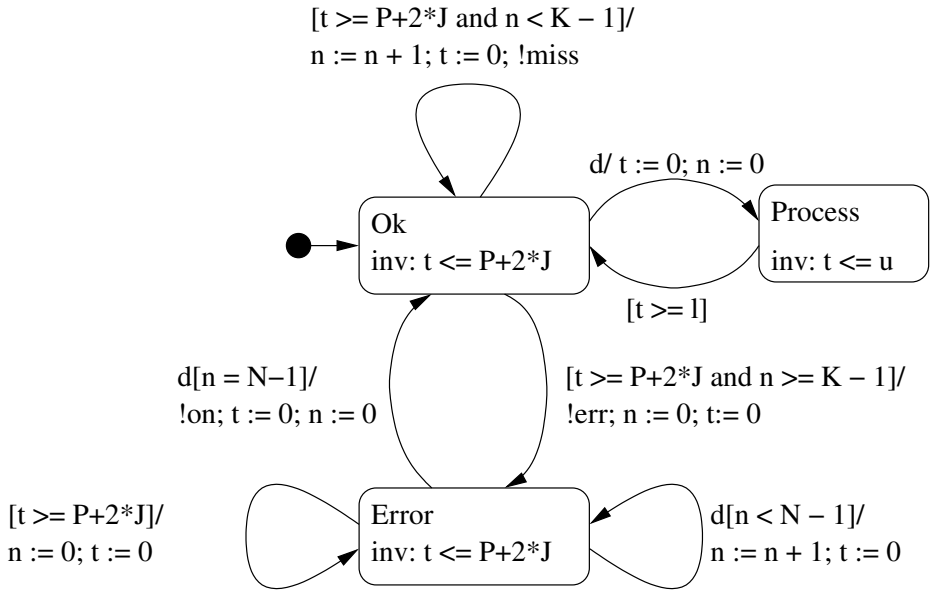


**Figure 7.5:** State machine of the message receiver

The next predicate specifies that the message receiver will only send an error signal $e$ if a timeout has occurred:

$$\mathrm{MR}_{s,2}(\theta) \overset{\text{def}}{\iff} \forall j : E(\theta_j) = err_s \implies \exists i,k : i < k \wedge k < j \wedge \\ \neg\,\mathrm{Error}(d_s, i)(\theta) \wedge \mathrm{TimeOut}(d_s, L, i, j)(\theta) \wedge \mathrm{Never}(err_s, k, j-1)(\theta) \quad .$$

Next, we define the validity and integrity of an *ok* signal. If the message receiver is in error state and it receives $N$ consecutive $d$ messages, then it will send an *ok* message within $\Delta_{ok}^{\mathrm{MR}}$ time units:

$$\mathrm{MR}_{s,3}(\theta) \overset{\text{def}}{\iff} \forall j : \mathrm{Error}(d_s, j)(\theta) \wedge \mathrm{Recover}(d_s, j)(\theta) \implies \\ \mathrm{AfterWithin}(ok_s, j, \Delta_{ok}^{\mathrm{MR}})(\theta) \quad .$$

The next predicate specifies that if an *ok* signal occurred, then the system was in an error state before and $N$ consecutive data messages had occurred, and no other *ok* signal has been emitted in between:

$$\mathrm{MR}_{s,4}(\theta) \overset{\text{def}}{\iff} \forall j : E(\theta_j) = ok_s \implies \exists i,k : i < k \wedge k < j \wedge \\ \mathrm{Error}(d_s, i)(\theta) \wedge \mathrm{occ}(d_s, N, i, k) \wedge \mathrm{Never}(ok_s, k, j-1)(\theta) \quad .$$

Next, the error logic component, to be specified later, has to be notified by any message receiver that did not receive a data message in time. This is indicated by a

*miss* message. We introduce this *miss* signal, 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 this is not allowed when trying to recover.

Only *one miss* signal is needed and not one per message receiver. The reason for this is that, in order to recover, *all* message receivers have to receive $N$ consecutive data messages. The presence of a *miss* signal indicates that there is one component which missed a data message during this period. Consequently, the error logic does not need to know which message receiver missed the signal.

A message receiver sends a *miss* message to the error logic whenever a data message has timed out from the data source *and* it is not in an error state. Again, sending a *miss* signal is delayed by $\Delta^{\mathrm{MR}}_{miss}$ time units.

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

If the message receiver is already in an error state, it will signal $N$ consecutive data messages using an *ok* message. Therefore, sending the *miss* signal is not necessary in this case. Also note that if we miss the $K$th data item at $j$, the $\mathrm{Error}(d_s, j)(\theta)$ predicate is true and no *miss* signal is emitted. Instead, following the assertion $\mathrm{MR}_{s,3}$, an $err_s$ signal will be sent. That is, it will not send both a *miss* signal and an $err_s$ signal.

The next predicate specifies the integrity of a *miss* event, that is, whenever a *miss* signal occurs the system was not in an error state before the occurrence of the *miss* signal, a data messages has not been received within the specified time, and no other *miss* signal has been emitted between the time-out of the data message and the occurrence of the *miss* signal.

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

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

**Lemma 7.1.** *For any message receiver s we have: if* $\forall i, j : \mathrm{TimeOut}(\mathit{miss}, NP + 2J, i, j)$ *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 7.2.** *If* $\mathrm{TimeOut}(\mathit{miss}, NP + 2J, i, j)$ *for all* $i, j$, *then* $\mathrm{Recover}(\{d_s \mid s \in S\}, i, j)$.

*Proof.* Follows directly from Lemma 7.1 and the definition of Recover [87]. □

Finally, we specify a message receiver for a source $s$ as

$$\mathrm{MR}_s(\theta) \overset{\text{def}}{\iff} \mathrm{MR}_{s,1}(\theta) \wedge \mathrm{MR}_{s,2}(\theta) \wedge \mathrm{MR}_{s,3}(\theta) \wedge \mathrm{MR}_{s,4}(\theta) \wedge \mathrm{MR}_{s,5}(\theta) \wedge \mathrm{MR}_{s,6}(\theta) \quad .$$

In the following section we formalise the interface specification of the error logic.

## 7.5.2 Error Logic

The error logic component accepts $err_s$ and $ok_s$ signals from each data source $s$. It also accepts a signal *miss* which indicates that there exists a data source $s$ that has not received data from its source during its 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 7.6.

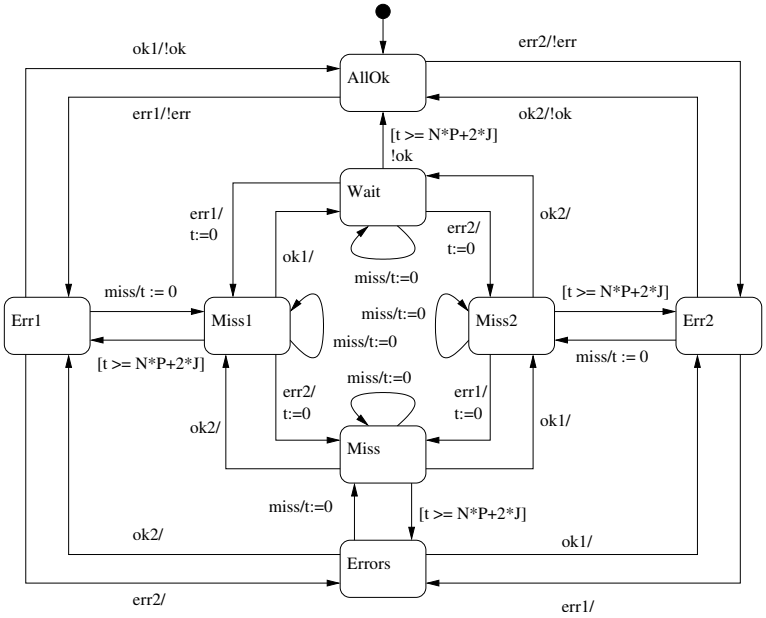


**Figure 7.6:** State Machine of the error logic component

In this state machine, the state AllOk indicates that the system is in the mode of normal operation. If the system is in this state and receives an $err_i$ signal from the message receiver $i$, then it moves to the state $err_i$ and signals an error by sending an *err* signal. The system may recover if it receives an $ok_i$ signal or it may receive an $err_j$ signal for $i \neq j$. That the system has to recover from an error from both data sources is indicated by staying in the state Errors. Moving to Errors does not require sending an

*err* signal, because we have already done so by taking a transition from AllOk to the state $err_i$ for some $i$. The system signals its recovery by sending an *ok* signal if it has received an $ok_i$ signal while in the state $err_i$.

As long as the system is in the AllOk state it ignores all *miss* signals. If a data source $i$ failed to send $K$ consecutive data messages to its message receiver, this will be signalled by the message receiver with a $err_i$ signal.

If the system is in an error state, that is, in one of the states $err_1$, $err_2$, or Errors, and it receives a *miss* signal, the error logic has to record that it has to wait for a time out of the miss signal and the required number of *ok* signals in order to return to normal operation. This fact is recorded in either the state $miss_1$, $miss_2$, *miss*, or Wait. These states indicate that the system has to wait for a time-out of the *miss* signal and for either an $ok_1$, $ok_2$, both, or no *ok* signal from the message receivers in order to recover. The state Wait is entered, whenever one of the $miss_i$ states has been left after receiving the corresponding $ok_i$ 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. While staying in one of these states, the system measures the time elapsed since the latest reception of a *miss* signal using the clock $t$. Therefore it always resets $t$ if it receives a *miss* signal *or* a $err_i$ signal. Recall, that the message receiver will not send both a *miss* signal and an $err_i$ signal during the same cycle.

The Wait state moves to the AllOk state after a timeout of the *miss* signal by emitting an *ok* signal. Note that the Wait signal is reachable by the following scenario: The system starts in the AllOk state. It then receives an $err_1$ signal from message receiver 1. Having received this signal the state changes to $err_1$. During the next period it receives a *miss* signal from message receiver 2! This causes a change of the state to $miss_1$, indicating that it has to receive an $ok_1$ signal from message receiver 1 and has to wait that message receiver 2 has to receive $N$ 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 the error logic changes its state to Wait. Now the error logic has to wait another period in order to assert that message receiver 2 has received its $N$th data message, after which it may signal recovery.

We proceed by formalising the behaviour of the error logic component. The error logic component accepts the signals $err_1$, $err_2$, *miss*, $ok_1$, and $ok_2$ as inputs and sends the signals *err* and *ok* as outputs.

Whether the error logic knows that a source $s$ is in an error state at position $i$ of trace $\theta$ is indicated by the following predicate:

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

Whether the error logic is in an AllOk state at position $i$ of a trace can be computed

by the following predicate:

$$\text{AllOk}(i)(\theta) \overset{\text{def}}{\iff} \forall s : \neg\, \text{Error}(i, s)(\theta) \quad .$$

Sending the *err* signal after having received an $err_s$ signal from some message receiver $s$ is delayed by $\Delta_{err}^{\text{EL}}$ time units. Then the validity of an *err* signal indicating error is specified by:

$$\text{EL}_1(\theta) \overset{\text{def}}{\iff} \forall i : \text{AllOk}(i)(\theta) \wedge (\exists s : E(\theta_{i+1}) = err_s) \implies \text{AfterWithin}(err, i+1, \Delta_{err}^{\text{EL}}) \quad .$$

Its integrity is specified by:

$$\text{EL}_2(\theta) \overset{\text{def}}{\iff} \forall j : E(\theta_j) = err \implies \exists i : i < j \wedge \text{AllOk}(i)(\theta) \wedge (\exists s : (E(\theta_{i+1}) = err_s) \wedge \text{Never}(err, i+2, j-1)(\theta) \quad .$$

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

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

The validity of an *ok* signal indicating recovery of a component $s$ is specified by, where emitting the *ok* signal is delayed by $\Delta_{ok}^{\text{EL}}$ time units:

$$\text{EL}_3(\theta) \overset{\text{def}}{\iff} \forall i : \exists s : \text{Recover}(i, s)(\theta) \wedge (\forall s : \neg\, \text{Error}(i, s')) \wedge (\exists k : \text{TimeOut}(miss, NP + 2J, k, i)(\theta)) \implies \text{AfterWithin}(ok, i, \Delta_{ok}^{\text{EL}}) \quad .$$

Observe that the assertion $\text{Recover}(i, s)(\theta) \wedge (\forall s' : \neg\, \text{Error}(i, s'))$ states that the message receiver $s$ is the last message receiver to recover.

The integrity of the *ok* signal is specified by:

$$\text{EL}_4(\theta) \overset{\text{def}}{\iff} \forall j : E(\theta_i) = ok \implies \exists i : i < j \wedge (\exists s : \text{Recover}(i, s)(\theta)) \wedge (\forall s : \neg\, \text{Error}(i, s)) \wedge \text{Never}(ok, i+1, j-1) \wedge (\exists k : \text{TimeOut}(miss, NP + 2J, k, i)(\theta)) \quad .$$

The error logic is then specified by the assertion:

$$\text{EL}(\theta) \overset{\text{def}}{\iff} \text{EL}_1(\theta) \wedge \text{EL}_2(\theta) \wedge \text{EL}_3(\theta) \wedge \text{EL}_4(\theta) \quad .$$

## 7.6 Correctness of the decomposition

To show the correctness of the decomposition of the global system into a receiver for each data source and an error logic, we first define general signals with a source identity and process identities $S$.

We observe that the internal signal are $I \stackrel{\text{def}}{=} \{ok_s, err_s, miss \mid s \in S\}$ and that the externally observable signals are $\{d_s, err, ok \mid s \in P\}$. The statement of correctness then is:

**Theorem 7.3.** $((\parallel_{s\in S} \mathrm{MR}_s) \parallel \mathrm{EL}) - I \implies \mathrm{TDS}$.

To establish this theorem, we first need to establish a sequence of intermediate steps, where we established the validity of each part of the TDS assertion individually. The proofs in PVS can be found in [87].

**Lemma 7.4.** $\Delta^{\mathrm{MR}}_{err} + \Delta^{\mathrm{EL}}_{err} \leq \Delta_{err} \wedge (\forall s : \mathrm{MR}_{s,1}(\theta)) \wedge \mathrm{EL}_1(\theta) \implies \mathrm{TDS}_1(\theta)$

The proof of this lemma is straight forward but tedious. The condition $\Delta^{\mathrm{MR}}_{err} + \Delta^{\mathrm{EL}}_{err} \leq \Delta_{err}$ is needed to establish that the composed system sends the *err* signal in time.

**Lemma 7.5.** $\Delta^{\mathrm{MR}}_{err} + \Delta^{\mathrm{EL}}_{err} \leq \Delta_{err} \wedge (\forall s : \mathrm{MR}_{s,2}(\theta)) \wedge \mathrm{EL}_2(\theta) \implies \mathrm{TDS}_2(\theta)$

**Lemma 7.6.** $\Delta^{\mathrm{MR}}_{ok} + \Delta^{\mathrm{EL}}_{ok} \leq \Delta_{ok} \wedge (\forall s : \mathrm{MR}_{s,3}(\theta)) \wedge \mathrm{MR}_{s,5}(\theta)) \wedge \mathrm{EL}_3(\theta) \implies \mathrm{TDS}_3(\theta)$

In the proof of this lemma we use Lemma 7.1 in conjunction with the assumption $\forall s : \mathrm{MR}_{s,5}(\theta)$ to establish that each data source has $N$ occurrences of a data signal from each component. The assumption $\forall s : \mathrm{MR}_{s,3}(\theta)$ asserts that a valid $ok_s$ signal is sent. The condition $\Delta^{\mathrm{MR}}_{ok} + \Delta^{\mathrm{EL}}_{ok} \leq \Delta_{ok}$ is needed to establish that the *ok* signal occurs in time.

**Lemma 7.7.** $\Delta^{\mathrm{MR}}_{ok} + \Delta^{\mathrm{EL}}_{ok} \leq \Delta_{ok} \wedge (\forall s : \mathrm{MR}_{s,4}(\theta)) \wedge \mathrm{MR}_{s,6}(\theta)) \wedge \mathrm{EL}_4(\theta) \implies \mathrm{TDS}_4(\theta)$

## 7.7 Conclusions

We have presented a compositional framework for the compositional verification of high-level real-time components which communicate by means of asynchronous signals. This framework reflects the full formal proof given in PVS [87], and is the result of a long and arduous path leading to consistent specifications of the parts and their properties. Compositional proof rules for parallel composition and hiding have been proved sound in PVS. The framework is based on timed event traces which are abstractions of the runs of the semantics of timed UML state machines [149]. In this way, we can use deductive verification in PVS to prove the correctness of a decomposition of a system into a number of asynchronously 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 provided by the Netherlands National Aerospace Laboratory. This case study has been supplied in the form of UML models. 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.

The compositional verification of the case study presented here 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.

Because we started with a specification, which was monolithic and contained errors, a redesign of the original system was necessary to enable the application of compositional techniques and to help improving our understanding of the model. Interestingly, this led to a design that is more flexible, for example, for changing the error logic, and more easily extensible, for example, to more data sources, than the original model.

We have found errors in the decomposed specification using model checking (we used the IF validation environment [16] and UPPAAL [91]) and by the fact that no proof in PVS could be found for the original specification. One of these errors was that we did not include a *miss* signal, which is needed to correctly observe recovery in the error logic component. This allowed the recovery of the system 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.

One final conclusion from this example is that, for specifications of the complexity of the MARS case study, especially with respect to its error logic, confidence in its correct functioning requires tool-based verification.