

Formal Verification of a Sensor Voting and Monitoring UML Model*

Christian Mrugalla³, Oliver Robbe², Ingo Schinz¹, Tobe Toben², and Bernd Westphal²

¹ OFFIS, Escherweg 2, 26121 Oldenburg, Germany, schinz@offis.de

² Carl von Ossietzky Universität Oldenburg, 26111 Oldenburg, Germany, {oliver.robbe,toben,westphal}@informatik.uni-oldenburg.de

³ Dipl.-Inform. Ch. Mrugalla, Hufeisen 60, 26605 Aurich, Germany, christian@mrugalla.info

Abstract. We report on the formal verification of a triple redundancy sensor voting and monitoring model written in UML using the UML Verification Environment (UVE). The original model as provided by Israel Aircraft Industries, Ltd. (IAI) doesn't adhere to the UVE modelling guidelines and is large in terms of the number of objects and the amount of concurrency compared to the capabilities of the UVE prototype. We describe how we were although able to apply the UVE and identified a potential mismatch between a requirement provided by IAI and the model, employing a number of significant abstractions.

1 Introduction

For the development of safety critical systems there is a growing interest in model-based approaches. This trend of model-based software development is supported by the recent specifications of the Object Management Group (OMG), in particular the Model Driven Architecture or version 2.0 of the Unified Modelling Language (UML) focusing on executable models.

In a *formal* model-based development process, there is typically a model of the system under construction given in a formalism with precise semantics which is complemented by (functional) requirements stated using a visual formalism like Live Sequence Charts [3] or plain temporal logic, e.g. CTL over terms of the model. In order to use the model to automatically generate the final implementation or to use the model as a *golden device*, i.e. as the reference against which manual implementations – possibly conducted by external suppliers – are compared, it is desirable to formally verify that the model satisfies the requirements.

The UML Verification Environment (UVE) [13] is an effort in this direction as it provides formal verification of high level models specified in UML. It employs finite state model checking to an appropriate representation of the behaviour of a given executable UML model using the Rhapsody semantics of UML [6, 4] as there is no universal formal UML semantics yet.

In an executable UML model, the structure is given by a class diagram and the behaviour has to be completely specified by state-machines and methods. The case-study discussed in this paper, for example, uses a subset of C++ as action language.

* This work was partly supported by the European research project IST-2001-33522 OMEGA and by the German Research Council (DFG) within project USE (DA 206/7-3) as part of the priority program “SoftSpez” (SPP 1064).

If a requirement is found not to hold for a given executable UML model, UVE generates a counter-example, i.e. a run of the model that doesn't adhere to the requirement. Counter-examples are presented in terms of the UML model employing a sequence- and a timing diagram to aid the designer in first understanding why the obtained run actually is a counter-example and second to adjust the model or the requirement, as both may contain errors.

In this paper we report on a case-study we conducted using a UML model provided by Israel Aircraft Industries, Ltd. (IAI) in their role as industrial partner of the IST OMEGA project. In this article, the focus is on the application of particular abstractions to yield a model that is treatable with UVE, in contrast to [13] where the case-study is only briefly mentioned.

While model checking of a particular requirement against a particular model is principally fully automatic, it is often impractical within a given amount of time and space. Large models become treatable by (manually) constructing abstractions, a process which still requires significant amount of expertise. The strategy for constructing abstractions that we present in the following is a first step towards a more general methodology as we expect most of the abstractions to apply (in particular instantiations) to general UML models of safety critical systems.

A work close to ours is [14], where an Executable UML (xUML) [9] model of a robot controller is verified using the UML model checker ObjectCheck [16]. The sensor voting system considered in this paper is different in that it uses an amount of UML outside the scope of xUML and therefore requires the particular abilities of UVE.

The paper is structured as follows. Section 2 briefly introduces the UML Verification Environment (UVE) and Sec. 3 outlines the subject of the case-study, the triple redundant sensor voting system in the version obtained from the supplier IAI. This version is not compliant to UVE and it is too large to be directly checked with UVE in a reasonable amount of space of time.

Section 4, the main contribution of this paper, shows how we were nevertheless able to apply UVE to a particular requirement on the case-study and discovered a potential mismatch between the model and one of the requirements provided by IAI along with the model. We in particular discuss how the specific approach presented in this section might generalise to general UML models of critical systems. Section 5 concludes and discusses further work.

2 The UML Verification Environment

With the UVE tool, requirements on a UML model can be specified and proven using formal verification. In contrast to testing, formal verification considers all possible runs of a system, hence being especially adequate in the domain of critical systems development. UVE uses the VIS [1] model checker, a tool for automatic formal verification of finite-state concurrent systems and is integrated in the UML design tool "Rhapsody in C++" offered by the company I-Logix. The requirements to be verified as well as potentially resulting counter-examples are described on the level of the UML model. That is, requirements may refer to

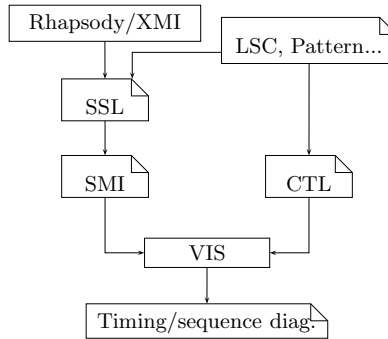


Fig. 1. UVE architecture. Arrows represent data-flows, typically through multiple tools or stages. Only the most important tools are shown.

variables, states, or events of the UML model. If there exists a run violating the requirement a counter-example is generated and presented as a sequence- and a timing diagram referring again to elements of the UML model and so allowing the animation and retracing of the counter-example in the UML model.

UVE follows a translational approach converting the UML model into the input language of the model checker. The structure and behaviour of the model are translated into the proprietary high-level intermediate language SSL (cf. Fig. 1) and successively transformed into an SSL representation without scopes, functions, or classes which translates directly into another proprietary intermediate language, SMI, that is basically Dijkstra’s guarded commands with rich type system and expression language. SMI is translated into a special kind of finite state machines used as the VIS model checker input. During the transformation several optimisations are conducted. For example the mode of variables and associations is set to constant if a static analysis determines that their value is never changed, or statements determined not to be relevant for a requirement are removed (cone-of-influence reduction).

There are two flavours of UVE differing in the way the UML model is obtained from the design tool before the translation. (R)UVE uses the Rhapsody API and the generated C++ code, and (X)UVE uses the exported XMI [12] representation thereby offering the possibility to be adapted to other UML design tools. Unfortunately there is currently no consensus on the use of XMI with the effect that XMI is in most cases not exchangeable between tools by different vendors. UVE currently supports the XMI generated by Rhapsody.

UVE supports different kinds of verification tasks. The simplest tasks, the drive-to-state or drive-to-property tasks, are for example very useful to automatically check whether a given state or a given configuration of attributes is possibly reachable and to obtain a counter-example. The appropriate properties are defined in terms of C++ expressions. General safety and liveness requirements of the model can be stated using the specification formalisms OFFIS patterns [10]. OFFIS patterns are stencils for temporal logical formulas. The most powerful specification language supported by UVE are LSCs [3], a kind of sequence diagrams of strictly larger expressive power that has a full formal semantics [7]. LSCs, for example, provide means to require liveness along livelines.

To obtain the static structure and the dynamic behaviour of a UML model UVE considers class diagrams and state-machine diagrams. All other diagrams are ignored. The action language for state-machine annotations and method implementations is a subset of C++.

Dynamic creation, deletion, and recreation of objects is supported if the number of objects alive at a single point in time is bounded by a user-given limit.

In class diagrams, UVE supports polymorphism (single inheritance and virtual methods) and bounded associations, aggregations, and compositions. Currently not supported are multiple inheritance and template classes.

In state-machine diagrams, UVE allows hierarchical state-machines, and- and or-states, triggered operations, and pseudo-states. Further supported are parameterised events and event hierarchies by inheritance. Time events and event deferring are currently not allowed; event queues have to be statically bounded. Recursion and loops are accepted when statically bounded. Integer, boolean, and pointer-to-object are allowed as types; floating-point values, arrays, and general pointers are not supported. Note that this enumeration refers to the capabilities of UVE at time of conducting the case study. In the meantime UVE has further evolved, e.g. it supports arrays in the current version and there are efforts to support infinite UML-Models in UVE [15].

For a complete description of features and restrictions of UVE the reader is referred to [13] and [11].

UVE is more appropriate for this case study than other tools, e.g. like ObjectCheck [16], since it supports a larger subset of UML, e.g. enhanced object oriented features like virtual methods and triggered operations, and a more convenient state-machine language comprising composite states.

3 The Sensor Voting UML Model

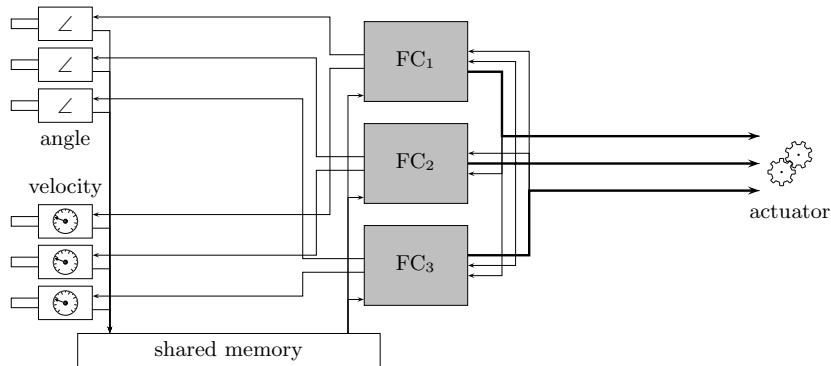


Fig. 2. Flight control computers and their control relation to the physical sensors and actuators deployed in aircraft. The lines from flight control computers to sensors indicate which sensor is triggered by which flight control computer. A bold line from a flight control computer to the actuator indicates the channel written to by this flight control computer.

The investigated sensor voting system [8] is taken from the domain of avionics. It models a part of the flight control system that periodically obtains parameters of the aircraft from sensors and computes commands for servo actuators (cf. Fig. 2). As the sensor voting system – comprising the sensors, the flight control computers, and the channels to the actuators – is safety critical, a very high reliability is required. To this end, the sensors for each aircraft parameter, the channels to the actuators, and the flight control computers are implemented triple redundant. The instance of a sensor voting system shown in Fig. 2 reads sensors for two kinds of parameters, namely an angle and a velocity, and commands a single actuator.

Sensors are operated by the sensor voting system in the following phases.

acquire: each flight control computer triggers one sensor of each kind of parameter to conduct a measurement and to write the measured value to the shared memory, e.g. in Fig. 2, FC₃ triggers the bottom-most angle and velocity sensors;

sample: each flight control computer reads the values of *all* sensors and all kinds of parameters from the shared memory, i.e. in Fig. 2, each flight control computer reads three values for the angle and three values for the velocity;

vote: each flight control computer decides whether a sensor’s value is considered to be correct or faulty based on a comparison of the three values read in the last phase. That is, it is principally possible that one flight control computer considers a sensor correct while another one considers it failed, e.g., if both receive different values due to transmission errors when reading from the shared memory.

Two sensors are considered correct if the difference between their measured values lies within a fixed tolerance. The third sensor is considered correct if the difference of its measured value lies within a fixed tolerance to any of the two other sensors.

monitor: each flight control computer stores the result of the last voting using one monitor for each sensor. The overall number of failures is counted and if a sensor fails more often than a fixed maximum, it is marked as out of order.

compute: each flight control computer computes the arithmetic average of the values sampled by all sensors that are considered correct and are not marked as out of order. Thus there is one final value per kind of parameter in each flight control computer. If no sensor is considered correct, an error value is the result of the compute phase, but this error value is actually not handled in the concrete model.

After these phases, each flight control computer computes an actuator command from the final values for each parameter kind, e.g. from the angle and the velocity in Fig. 2, and writes it to its channel, i.e. the connection to the actuator.

Each flight control computer votes and monitors the three channels based on the command values as read back from the channels, i.e. each flight control computer has an individual perception of each channel’s health. The effect of the channel voting lies, in contrast to the sensors, outside the scope of the model, i.e. the result is actually not used in the model.

An (*execution*) *cycle* of a flight control computer comprises the operation of sensors in the phases described above, the computation of the actuator command, and the voting and monitoring of channels; the sensor voting system periodically executes cycles.

Figure 3(a) shows an object diagram of the model. For readability, it shows only a single flight control computer, instead of all three, with its two sensor voting modules and one channel voting module.

The cycles of a flight control computer as explained above are implemented by the objects of classes *SysRun*, *SysModel*, and *FC*. The cycles in *SysModel* are triggered by *SysRun* that in turn is triggered by an associated object of class *RTC* representing a real-time clock. The actual implementation of the cycle by a state-machine is discussed below.

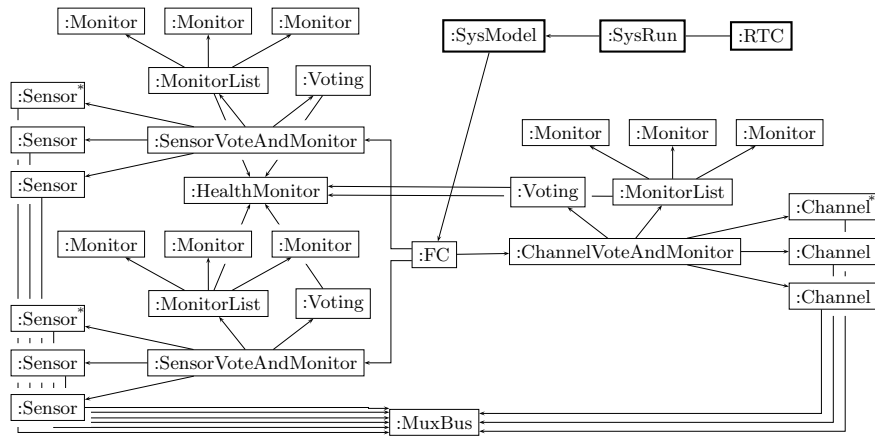
The class *SensorVoteAndMonitor* provides methods for all phases on sensors as introduced above. Each object of this class is responsible for one parameter of the aircraft, for example, in Fig. 3(a) the upper one could be responsible for the angle and the lower one for the velocity. Each *SensorVoteAndMonitor* references three sensors and (indirectly) owns a set of *Monitor* objects and an object of class *Voting* that encapsulates the voting and monitoring functions. All monitoring and voting objects of an *FC* share an object of class *HealthMonitor* to keep track of sensor failures. This is where sensors and channels may be marked to be out of order.

In the example, the *FC* is connected to two *SensorVoteAndMonitor* objects thus measuring two parameters of the aircraft as assumed in Fig. 2. This is no inherent restriction of the model, it is designed to be adapted to any number of aircraft parameters.

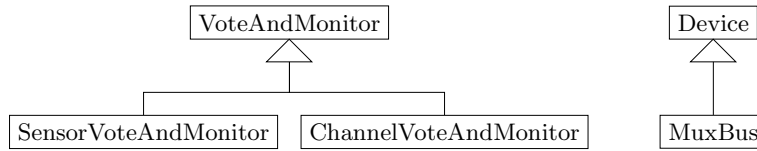
Figure 3(b) shows the most relevant inheritance relations in the model in a class diagram. As voting and monitoring is the same for sensors and channels, there is a class *VoteAndMonitor* that is specialised for sensors and channels. The specialisation for sensors, for example, adds a method for computing the arithmetic average value of the correct sensors.

The class *Device* provides an abstract interface for reading and writing shared memory using events and virtual methods and is specialised by *MuxBus* that corresponds to the shared memory shown in Fig. 2.

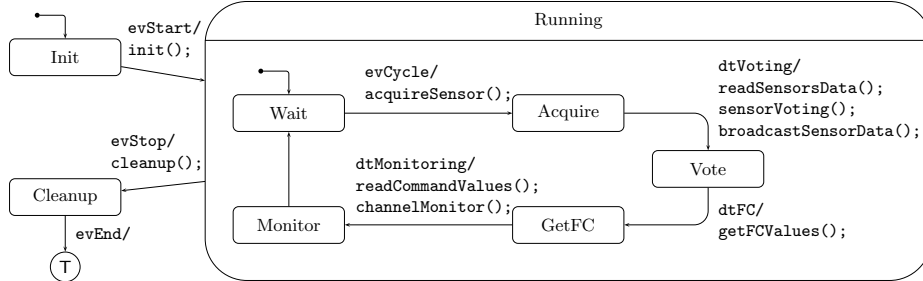
The central state-machine, belonging to *SysModel*, is shown in Fig. 3(c). It implements the execution cycle as introduced above, i.e. once a new cycle is started (triggered by an object of class *RTC* modelling a real-time clock) by event *evCycle*, all sensors are acquired. After a fixed number of ticks from the *RTC*, the *SysRun* sends an event *dtVoting*. The acquired values are then read from the shared memory, the sensors are voted and monitored, and the resulting arithmetic average values are transferred to the *FC*. After another fixed number of ticks, *SysRun* sends an event *dtFC* on which the command value is obtained from the *FC* and written to the channel controlled by this *FC*. After another fixed number of ticks, *SysRun* sends an event *dtMonitoring* on which the channels are voted.



(a) **Object diagram** of one flight control computer and the associated voted and monitored sensors and channels. The sensors acquired and the channel written to by this flight control computer are marked with an asterisk (*). The three objects of classes *SysModel*, *SysRun*, and *RTC* are active.



(b) **Inheritance relations** in the sensor voting model.



(c) **The state-machine** of class *SysModel* controls the cycle of sensor acquiring, sampling, and voting and monitoring, the transfer of values from the flight control computer to its channel, and finally the voting and monitoring of channels. Event *evCycle* is (indirectly) sent by the *RTC* object. Events like *dtVoting* are generated by the *SysRun* object after a fixed number of ticks from the *RTC*. That is, there is a fixed amount of *RTC* ticks assigned to each phase of the behaviour.

Fig. 3. Parts of the original sensor voting model.

Other classes with state-machines are *Monitor* with three states, *SysRun* with a state-machine similar to *SysModel* as it is able to control a number of additional, non-monitored sensors, *RTC* with two states, and *Device* with three states. The remaining behaviour, e.g. the voting algorithm, is provided as methods written in C++.

IAI provided a fully executable UML model of the sensor voting system as described here written with I-Logix’ “Rhapsody in C++”. Being fully executable means that Rhapsody generates C++ code that, if compiled and linked against an event-handling framework provided by I-Logix, yields a runnable binary. The Rhapsody code generator can *instrument* the code s.t. the binary connects at runtime to a running Rhapsody. Rhapsody then graphically traces, e.g., in which state each state-machine in the binary is and thereby provides an animation of the model.

4 Formal Verification of the Sensor Voting UML Model

The aim of the case-study we report on in this paper was to apply UVE in order to formally verify a set of crucial requirements for the sensor voting system as presented in Sec. 3. The requirements have been provided by IAI in form of natural language text along with the model. One of them is the correctness of the voting algorithm or, more precisely, that if just after the voting phase (cf. Sec. 3) there is at least one sensor of one kind of parameters voted correct and not out of order, then there is at least one sensor of this kind of parameters which is voted to be correct and whose value deviates at most $\delta/2$ from the computed arithmetic average value, where δ is the sensor tolerance. Figure 5 gives a formalisation of this requirement.

In order to verify this requirement with UVE, there are two obstacles. First of all, the UML model has to comply to UVE’s modelling guidelines that restrict the use of certain features of UML and the C++ language (cf. Sec. 2). Some of the restrictions given by the modelling guidelines arise since they are considered to be out of scope for UML model checking, e.g. services of the underlying operating system like file I/O operations. Other restrictions are given since they allow for more compact representations of the model, e.g. disallowing pointer arithmetics (even in arrays) is a basis for UVE’s representation of objects. A third category of restrictions stems from the prototype status of UVE. Some features of UML, like timeout events and arrays, were simply not supported by the early UVE version that has been used for the case-study.

The second obstacle is the well-known state explosion problem of model checking. That is, the complexity of conducting a model checking task, e.g. to prove the requirement given above, grows in the worst case exponentially with the number of objects in the system. The number of active objects in the system additionally raises the complexity.

Only by looking at the object diagram in Fig. 3(a) one can estimate that the model is large and that it is expensive (in terms of space and time) to verify it unless significant reductions take place, since Fig. 3(a) shows only the objects of *one* flight control computer of which we have three in the full model.

There are a number of automatic techniques to reduce the complexity that are typically applied first, the most prevalent being cone-of-influence reduction [2] that computes from the model description those variables that can possibly influence a given requirement. All other variables, or attributes, can safely be removed. Unfortunately this technique doesn't apply well to UML models since the event queue shared by nearly all objects has the consequence that this static technique usually has the result that all objects possibly influence all other objects and thus nothing is removed from the model.

A technique specific for UVE is the *crystallisation* of aggregations [13]. That is the pre-computation values of those links that can statically be identified to be constant during runtime. These links are then explicitly turned into constants and hence don't contribute to the model checking complexity any more.

Similarly, the initial step of the system where all objects are created and started can be pre-computed. After the initial step, many links and attributes are never written again and can also be turned into constants.

If the reductions obtained by the automatic techniques is insufficient, the model can manually be reduced. For example by choosing more efficient encodings or by removing redundancy from the model.

For the sensor voting model this is still not sufficient. So we constructed a non-exact abstraction, i.e. a modification of the model s.t. the behaviour of the modified model *includes* the behaviour of the original one. Being able to establish a requirement on the abstract model then implies that the requirement also holds for the original model while obtained counter-examples may be spurious. Although, this is decidable in bounded models by simply replaying the counter-example using the animation. If the counter-example is replayable, then it is not spurious. And then it gives valuable hints about possible bugs in the design.

When applying abstractions manually, as we have done in this case-study, the results of UVE have to be taken with great care since in addition to UVE itself not being certified, there is the risk that additional errors are introduced into the model when constructing the abstraction. If a requirement can be established for the abstract model, then it actually only holds for the concrete model *assuming* that the abstraction has been properly constructed and no new errors have been introduced. To complete a proof of a requirement, this assumption has to be discharged in addition to the original requirement.

Nonetheless this case-study shows the relevance of realisations of formal verification techniques for UML. First of all, we have discovered a non-obvious discrepancy between the behaviour of the original model and the requirements specification, showing UVE to be valuable as a *debugging* aid. Secondly, some of the abstractions show potential for automatisation, for example the detection of some redundancies. An automated procedure is less error-prone and can be certified to yield correct abstractions by construction.

The rest of this section is structured as follows. In Sec. 4.1 we discuss in some detail how UVE compliance has been achieved in order to give a feeling for the power of UVE in spite of the substantial catalogue of restrictions. Most of these modifications of the model are independent from the studied requirement.

In Sec. 4.2 we provide the formalisation of the requirement given above and elaborate on the (non-exact) abstractions we have applied in order to be able to use UVE. These modifications depend on the requirement in that they, for example, exploit that the considered requirement doesn't refer to the channel voting.

By applying UVE to the abstract model we actually discovered a discrepancy between the sensor voting model and the requirement on the voting algorithm as given above. This is discussed in Sec. 4.3. Section 4.4 gives approximate figures on the runtimes for the particular verification task from Sec. 4.3.

Note that we actually used a number of different versions of the sensor voting model that have been continuously negotiated with the provider. The one presented in Sec. 3 is the most sophisticated. As model checking experiments have been conducted on many intermediate versions, the strategy we elaborate on in the following is the essence of all these efforts: if the case-study started anew today with the model from Sec. 3, results with UVE would be obtained applying the following strategy.

4.1 Achieving UVE Compliance

The full model doesn't adhere to UVE's modelling guidelines that restrict the use of certain features of UML and the C++ language (cf. Sec. 2).

The major issues in the example were the use of file I/O for logging and configuration purposes, floating point arithmetics in the computation of arithmetic averages, and the use of integer arrays, that the UVE version used for the case study did not support.

File I/O for logging purposes has completely been removed from the model since it doesn't contribute to the behaviour of the model to be checked. File I/O for configuration purposes, i.e. the possibility to use a file to select between configurations of non-voted sensors and to feed a sequence of sensor values to the model, has been eliminated by hardcoding a configuration of non-voted sensors that configures no non-voted sensor since the requirement doesn't refer to non-voted sensors. Concerning sensor values we employ a feature of UVE that allows to consider certain attributes as *inputs*. The model checker then examines all possible values. Obtaining sensor values using inputs instead of reading them from a file is the appropriate representation, since when model checking the system one indeed wants it to be checked for all possible combinations of sensor values.

Floating point arithmetics has been replaced by a manual discretisation s.t. we finally use only integer types. Internal arrays of, e.g., the *MuxBus* were replaced by a newly introduced class *ValueArray* that implements array functionality using only the subset of C++ supported by UVE.

4.2 Treating Complexity by Abstraction

From the perspective of model checking UML models, the compliant sensor voting model is *large* since it comprises

- about 74 objects of user-defined classes, where inherited parts also count as objects due to the representation of inheritance in UVE, e.g. a single *MuxBus* object counts as two objects: one *Device* and one *MuxBus*;
- 12 objects have state-machines that consume 12 different kinds of events,
- four active classes, the three explicit ones *RTC*, *SysModel*, and *SysRun* plus a global one from the Rhapsody framework whose thread all remaining objects are assigned to; this results in ten active objects at runtime of the full model;
- significant amounts of arithmetics on sensor values, e.g. computing the arithmetic average which involves floating point types;
- large arrays, e.g. the *MuxBus* employs an array of 1024 integers; and
- some data is stored redundantly in different attributes.

As explained in the introduction to this section, the model was (expectedly) found to be far too large to be directly treatable with UVE even with the complexity reductions automatically applied by UVE.

The manual removal of complexity from the model in order to obtain results in a given amount of time and space was conducted with the aim to keep the changes minimal and simple, in particular to be able to track changes in the original model also in the abstract model

One simple abstraction is the elimination of redundancies and to make the domains of all types as small as possible. The correctness of such model transformations is more or less obvious. A standard adjustment is the choice of the right domain for integers. Models typically don't use a full range of 32 or 64 bit. For the sensor voting model, a range of $[-8, 7]$ has been used. When introducing the new class *ValueArray*, the array sizes have been adjusted to a minimal amount, e.g., no longer using 1024 places in the *MuxBus*. Additional smaller changes comprise, for example, that some attributes could manually be identified to actually be constant.

A next step addresses behaviour that does obviously not contribute to the requirement. For example, most behaviour for system shutdown has been removed as we consider the system to be reactive and non-terminating. Thereby a number of events for system shutdown became obsolete. Similarly, the channels and their voting and monitoring can be removed from the model since they don't affect the studied requirement. This leads to both a leaner structure and behaviour.

Furthermore, some behaviour can be encoded more efficiently. For example, the state-machine of class *SysModel* has been modified to react directly on events of the new type *evTick* modelling passing time. *evTick* events are declared to be sent by the environment at any point in time. In other words, the model checker is able to notify the *SysModel* any time about passed time, thereby abstracting from the particular timing in the executable model. Since we use a configuration without non-voted sensors, the objects of class *SysRun* can be removed if the actions that trigger the computation of the arithmetic average is moved to *SysModel*.

Evidently all these steps are not sufficient to treat the major cause for complexity in the example, namely the sheer amount of objects and links. Therefore the model has been *focused* to the object diagram shown in Fig. 4(a). As the requirement applies to all flight control computers and all kinds of aircraft pa-

```

let
  s := root → p_System → itsSensors,
  m := s → itsMonitorList,
  d := root → p_System → itsDevice
in
  root→p_System→IS_IN(Done) ∧ (m→itsMonitors[0]→IS_IN(Ok)
    ∨ m→itsMonitors[1]→IS_IN(Ok)
    ∨ m→itsMonitors[2]→IS_IN(Ok))
  ⇒ (|s→finalData - d→memBuffer[1]| ≤ δ/2 ∧ m→itsMonitors[0]→IS_IN(Ok)
    ∨ |s→finalData - d→memBuffer[2]| ≤ δ/2 ∧ m→itsMonitors[1]→IS_IN(Ok)
    ∨ |s→finalData - d→memBuffer[3]| ≤ δ/2 ∧ m→itsMonitors[2]→IS_IN(Ok))
    
```

Fig. 5. Formalisation of the requirement as introduced in Sec. 4. It directly corresponds to a C++ expression that is used to instantiate the corresponding pattern from the OFFIS pattern library (cf. Sec. 2). Note that it is a *state invariant*, that is, an invariant that doesn't employ temporal modalities.

p_System denotes a top-level object of the model that is not part of any other object. In UVE expressions, top-level objects can be navigated from a (virtual) root object denoted by 'root'.

rameters independently, it is by symmetry [5] sufficient to consider only *one* flight control computer and within this one only *a single* kind of value, and thus a single set of sensors and voting and monitoring objects. The remaining flight control computers are not necessary to trigger their sensors, since sensor values are obtained as inputs, i.e. at the point in time when they are read. This has, in addition to a smaller number of objects whose attributes need not be represented, the effect that links can be encoded using types with smaller domains. Fewer objects simply need a smaller domain of identities.

The remaining major cause for complexity is the large number of active objects, i.e. the concurrency in the model. Recall that both classes *RTC* and *SysRun* have been removed from the model. So *SysModel* remains as the only active class in addition to the global thread all other objects run in. As the actions of *SysModel* are synchronised with the rest of the model, it need not remain active. The model with active *SysModel* is equivalent to a single thread model.

Finally we obtained a model structured similar to the one shown in Fig. 4 with about 16 objects, 8 of them with state-machine, and only 4 kinds of events all running in a single thread.

4.3 Verification Result

Applying UVE to the finally obtained abstract model yields that the requirement on the voting algorithm as formalised in Fig. 5 doesn't hold. The counter-example reveals the following non-obvious issue. Consider three sensors s_1, s_2, s_3 with values s.t. the distance between the value of s_2 and the values of s_1 and s_3 is

smaller than the tolerance,

$$d(v(s_1), v(s_2)) \leq \delta, d(v(s_2), v(s_3)) \leq \delta,$$

but s_1 deviates from s_3 by more than the tolerance,

$$d(v(s_1), v(s_3)) > \delta.$$

By the first two inequations, all sensors are voted correct. Now if s_2 has failed in the past and is currently considered to be out of order, then only s_1 and s_3 contribute to the arithmetic average. And both have a distance larger than $\delta/2$ from the arithmetic average. This violates the requirement stated in Sec. 4.2.

4.4 Verification Runtimes

Technically, we distinguish three phases of a verification task. Firstly, the *model generation*, i.e. compilation of the UML model into a finite state machine (FSM) palatable for the underlying VIS model checker [1], secondly, actual *model checking*, and thirdly – undoing optimisations of the VIS on the level of the FSM – the *completion and back-translation* of the counter-example to terms of the UML model.

For the focused sensor voting model, model generation is performed in the order of 15 min., model checking in the order of 90 min., and completion in the order of hours.⁴

The reason for the latter is not fully understood. We conjecture that the presence of the event queue and the asynchronous nature of UML models is responsible for the large overall verification time when compared to, e.g., FSMs of roughly the same size but stemming from a synchronous modelling domain like StateMate statecharts.

5 Conclusion

We have reported experiences from a successful case-study of model checking for UML using the UVE tool-set. Given a thorough understanding of the model and appropriate experience in the application of UVE, it is possible to formally verify invariants for a UML model of a safety critical system like the sensor voting system using a manually derived abstraction. The instances of the applied abstractions depend on the studied requirement, but we expect the strategy to apply to general UML models of safety critical systems.

Further work comprises better integration of the abstractions s.t. the abstract model is automatically obtained from the original by applying abstraction operations and to investigate the automatic application of abstractions like focusing on particular cases justified by symmetry [5].

References

1. Robert K. Brayton, Gary D. Hachtel, Alberto L. Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen A. Edwards, Sunil P. Khatri,

⁴ UltraSparc III+, 900 MHz, 2GB

- Yuji Kukimoto, Abelardo Pardo, Shaz Qadeer, Rajeev K. Ranjan, Shaker Sarwary, Thomas R. Shiple, Gitanjali Swamy, and Tiziano Villa. VIS: A system for verification and synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer, 1996.
2. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
 3. Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, July 2001.
 4. Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva. A discrete-time uml semantics for concurrency and communication in safety-critical applications. *Science of Computer Programming*, 55(1–3):81–115, March 2005.
 5. Werner Damm and Bernd Westphal. Live and let die: LSC-based verification of UML-models. *Science of of Computer Programming*, 55(1–3):117–159, March 2005.
 6. David Harel and Hillel Kugler. The rhapsody semantics of statecharts. In Hartmut Ehrig, Werner Damm, Martin Große-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkämper, editors, *Integration of Software Specification Techniques for Applications in Engineering*, number 3147 in LNCS, pages 325–354. Springer-Verlag, 2004.
 7. Jochen Klose. *Live Sequence Charts: A Graphical Formalism for the Specification of Communication Behavior*. PhD thesis, Carl von Ossietzky Universität Oldenburg, 2003.
 8. Israel Aircraft Industries, Ltd. OMEGA: Project case studies, IAI case study, 2004. <http://www-omega.imag.fr/cs/IAI/IAI.php>. Accessed 2005-09-17.
 9. Stephen J. Mellor and Marc J. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley, 2002.
 10. OFFIS. *Verification of State Machine Designs - Pattern Library*. User Manual.
 11. OFFIS. OMEGA: OMEGA toolset, UVE, 2004. <http://www-omega.imag.fr/tools/UVE/UVE.php>. Accessed 2005-09-17.
 12. OMG. Corba, xml and xmi resource page. <http://www.omg.org/technology/xml/>. Accessed 2005-09-17.
 13. Ingo Schinz, Tobe Toben, Christian Mrugalla, and Bernd Westphal. The Rhapsody UML Verification Environment. In Jorge R. Cuellar and Zhiming Liu, editors, *Proceedings of the 2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), Beijing, China*, pages 174–183. IEEE, September 2004.
 14. Natasha Sharygina. *Model Checking of Software Control Systems*. PhD thesis, The University of Texas at Austin, 2002.
 15. Bernd Westphal. LSC verification for UML models with unbounded creation and destruction. In Byron Cook, Scott Stoller, and Willem Visser, editors, *SoftMC 2005, Workshop on Software Model Checking (Satellite Workshop of CAV '05)*, ENTCS. Elsevier B.V., July 2005. To appear.
 16. Fei Xie. *Integration of Model Checking into Software Development Processes*. PhD thesis, The University of Texas at Austin, August 2004. UTCS Technique Report TR-04-29.