A Modelling and Simulation Based Approach to Dependable System Design

Miriam Zia, Sadaf Mustafiz, Hans Vangheluwe, and Jörg Kienzle

School of Computer Science, McGill University
Montreal, Quebec, Canada
{mzia2, sadaf, hv, joerg} @cs.mcgill.ca

Abstract. Complex real-time system design needs to address dependability requirements, such as safety, reliability, and security. We introduce a modelling and simulation based approach which allows for the analysis and prediction of dependability constraints. Dependability can be improved by making use of fault tolerance techniques. The de-facto example in the real-time system literature of a pump control system in a mining environment is used to demonstrate our model-based approach. In particular, the system is modelled using the Discrete EVent system Specification (DEVS) formalism, and then extended to incorporate fault tolerance mechanisms. The modularity of the DEVS formalism facilitates this extension. The simulation demonstrates that the employed fault tolerance techniques are effective. That is, the system performs satisfactorily despite the presence of faults. This approach also makes it possible to make an informed choice between different fault tolerance techniques. Performance metrics are used to measure the reliability and safety of the system, and to evaluate the dependability achieved by the design. In our model-based development process, modelling, simulation and eventual deployment of the system are seamlessly integrated.

1 Introduction

Model-based approaches are used to represent the structure and behaviour of systems, which are becoming increasingly complex and involve a large number of components and domain-specific requirements [1][2]. Dependable systems, in particular, must satisfy a set of functional requirements, and in addition, must adhere to constraints which ensure correct behaviour of the system. Safety, security and reliability are a few such dependability requirements. The necessity for accomplishing these constraints has spawned new fields of research. The most prominent area is that of fault-tolerant systems, and the introduction of fault tolerance design in the software development process is an emerging topic.

We are interested in developing the model-based process illustrated in Fig. 1 for designing a dependable system. The process allows us to predict the behaviour of a specific system, and compare it to the behaviour of a fault-tolerant implementation of the same system. This is done through a sequence of manual activities. First, from functional requirements, a model is derived which represents the structure of a chosen system. A fault injection mechanism is also modelled as a means to generate faulty behaviour of the system. Simulation results indicate how the system performs in the

L. Briand and C. Williams (Eds.): MoDELS 2005, LNCS 3713, pp. 217–231, 2005.

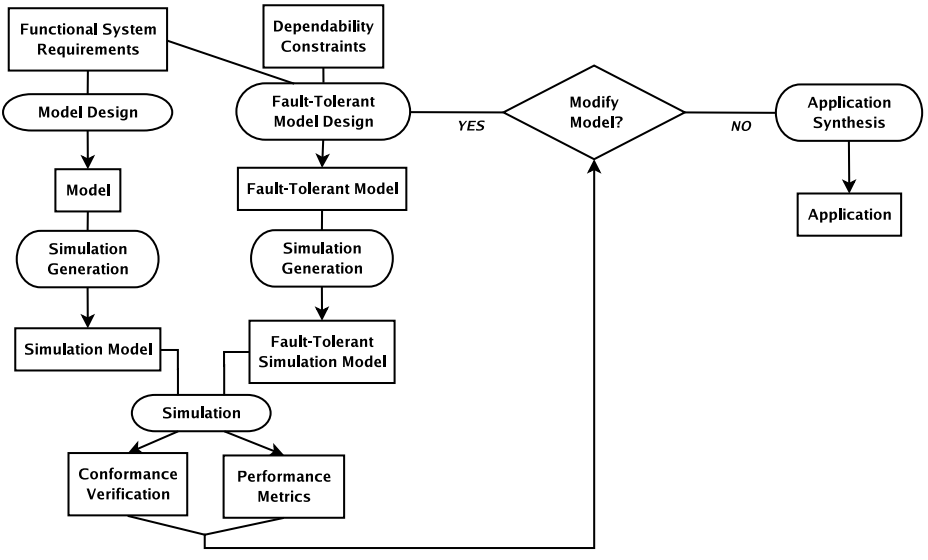

Fig. 1. The Model-based Process

presence of faults, and whether it conforms to the specified requirements. Secondly, from dependability constraints, a fault-tolerant model is created which includes techniques designed to improve on the initial system. A fault-tolerant simulation model is derived and simulated to gather performance data. This data reflects the dependability constraints that must be satisfied by the system.

Although research has been done in *formal modelling* and analysis of fault tolerance properties [3][4], either using natural language description of models, probabilistic models, figures of fault-trees or Markov models, we suggest using the formalism DEVS (*Discrete EVent System specification*). In our case study, the initial system as well as the fault tolerant system are translated into DEVS.

The paper is structured as follows. Section 2 presents essential background concepts relating to the DEVS formalism and to fault tolerance. Section 3 describes the real-time *Pump Control System* (PCS) chosen to demonstrate our process. We introduce its functional requirements and dependability constraints and briefly discuss why modelling and simulation is an appropriate approach, and why DEVS is an suitable modelling formalism. Section 4 introduces the model of the PCS, and the means by which fault injection is introduced in the system. A PCS failure situation is described in Section 5, and a fault-tolerant model is presented that counteracts this failure. Furthermore, safety and reliability are defined as the dependability constraints that are threatened by failure of the PCS. In Section 6, implementation-specific and experimental simulation framework details are outlined. Mathematical equations are presented to quantify the safety and reliability of the PCS, and results of the simulations are analyzed to compare the performance of the PCS in the two models. Finally, some general conclusions about our model-based process are drawn in Section 7.

2 Background

This section introduces the modelling formalism used in the case study, the *DEVS (Discrete EVent system Specification)* formalism and gives a brief overview of fault tolerance and the technique we apply in our work.

2.1 The DEVS Formalism

The DEVS formalism was introduced in the late seventies by Bernard Zeigler to develop a rigorous basis for the compositional modelling and simulation of discrete event systems [5][6]. The DEVS formalism has been successfully applied to the design and implementation of a plethora of different complex systems such as peer-to-peer networks [7], transportation systems [8], and complex natural systems [9]. In this section we briefly present the DEVS formalism.

A DEVS model is either *atomic* or *coupled*. An atomic model describes the behaviour of a reactive system. A coupled model is the composition of several submodels which can be atomic or coupled. Submodels have *ports*, which are connected by channels. Ports have a type: they are either *input* or *output* ports. Ports and channels allow a model to receive and send signals (events) from and to other models. A channel must go from an output port of some model to an input port of a different model, from an input port in a coupled model to an input port of one of its submodels, or from an output port of a submodel to an output port of its parent model.

An atomic model has, in addition to ports, a set of *states*, one of which is the *initial* state, and two types of transitions between states: *internal* and *external*. Associated with each state is a *time-advance* and an *output*.

Atomic DEVS [1]

An **atomic DEVS** is a tuple $(S, X, Y, \delta^{int}, \delta^{ext}, \lambda, \tau)$ where S is a set of **states**, X is a set of **input events**, Y is a set of **output events**, $\delta^{int} : S \to S$ is the **internal transition function**, $\delta^{ext} : Q \times X \to S$ is the **external transition function**, $\lambda : S \to Y$ is the **output function** and $\tau : S \to \mathbb{R}_0^+$ is the **time-advance** function.
In this definition, $Q = \{(s,e) \in S \times \mathbb{R}^+ \mid 0 \leq e \leq \tau(s)\}$ is called the **total-state space**, for each $(s,e) \in Q$, e is called the **elapsed-time**.[2]

Informally, the operational semantics of an atomic model are as follows: the atomic model starts in its initial state, and it will remain in any given state for as long as its corresponding time-advance specifies or until input is received on some port. If no input is received, when the time of the state expires, the model sends output as specified by λ (before changing the state), and subsequently jumps to the new state as specified by δ^{int}. On the other hand, if input is received before the time for the next internal transition expires, then it is δ^{ext} which is applied. The external transition depends on the current state, the time elapsed since the last transition and the inputs from the input ports.

The following definition formalises the concept of coupled DEVS models[3]

[1] For simplicity, we do not present a formalisation of the concept of "ports".

[2] $\mathbb{R}_0^+$ denotes the positive reals with zero included.

[3] For simplicity, this "formalisation" does not deal with ports, and it leaves out the proof of well-definedness for coupled models.

Coupled DEVS

A **coupled DEVS** named D is a tuple $(X, Y, N, M, I, Z, select)$ where X is a set of **input events**, Y is a set of **output events**, N is a set of **component names** such that $D \notin N$, $M = \{M_n \mid n \in N, M_n$ is a DEVS model (atomic or coupled) with input set X_n and output set $Y_n\}$ is a set of DEVS **submodels**, $I = \{I_n \mid n \in N, I_n \subseteq N \cup \{D\}\}$ is a set of **influencer** sets for each component named n, $Z = \{Z_{i,n} \mid \forall n \in N, i \in I_n . Z_{i,n} : Y_i \rightarrow X_n \; or \; Z_{D,n} : X \rightarrow X_n \; or \; Z_{i,D} : Y_i \rightarrow Y\}$ is a set of **transfer functions** from each component i to some component n, and $select : 2^N \rightarrow N$ is the **select** function.

Connectivity of submodels is expressed by the influencer set of each component. Note that for a given model n, this set includes not only the external models that provide inputs to n, but also its own internal submodels that produce its output (if n is a coupled model.) Transfer functions represent output-to-input translations between components, and can be thought of as channels that make the appropriate type translations. The *select* function takes care of conflicts as explained below.

The semantics for a coupled model is, informally, the parallel composition of all the submodels. This is, each submodel in a coupled model is assumed to be an independent process, concurrent to the rest. There is no explicit method of synchronization between processes. Blocking does not occur except if it is explicitly modelled by the output function of a sender, and the external transition function of a receiver. There is however a *serialization* of events whenever there are two submodels that have a transition scheduled to be performed at the same time. Logically, the transitions are assumed to be done in that time instant, but its implementation on a sequential computer is serialized. The coupled model has a *select* function which chooses one of the models to undergo the transition first.

2.2 Fault Tolerance

Complex computer systems are increasingly built for highly critical tasks, from military and aerospace domains to industrial and commercials areas. They are critical in the sense that their failures may have severe consequences ranging from loss of business opportunities, physical damage, to more catastrophic loss, such as human lives. Systems with such responsibilities should be highly *dependable*. A number of varied means of achieving this goal have been established and should be considered jointly during hardware as well as software development: *fault prevention*, *fault removal*, *fault forecasting* and *fault tolerance* [10]. In particular, we will discuss fault tolerance in more detail in this section.

The idea of incorporating means for fault tolerance in order to achieve system dependability has developed considerably since the original work by von Neumann in the mid-1950s [11], and many techniques have been established. To discuss fault tolerance more meaningfully, a definition of *correct system behaviour* is needed: the specification. As long as the system satisfies the specification, it is considered to be behaving correctly. A failure can then be defined as an observable deviation from the system specification. An error is that part of the system state that leads to a failure. The error itself is caused by some defect in the system; those defects that cause observable errors are called *faults* [12]. Fault tolerance aims at preventing failures in the presence of

hardware or software faults within the system. Therefore, as soon as an error has been detected, it must be corrected to ensure that a system continues to deliver its services and to avoid a potential failure later on in the execution.

These corrective measures need to be taken to keep the error from propagating to other parts of the system, thus preventing further damage. Once the error is under control, error recovery is applied and a correct error-free system state is restored. There are two basic recovery techniques [13]:

Backward error recovery replaces the erroneous system state with some previous correct state.

Forward error recovery attempts to construct a coherent, error-free system state by applying corrective actions to the current, erroneous state.

A popular form of forward error recovery is *Triple Modular Redundancy* (TMR). TMR uses three identical copies of a unit instead of one, and an intelligent, application-specific voting scheme which is applied to their output. In stateless cyclic systems, where one iteration of execution does not depend on the previous run, this mechanism allows for faults to be masked. This technique will be used in this case study to remedy the failure scenario discussed in section 5.1.

3 Modelling and Simulation Based Design: An Example

In Modelling and Simulation Based Design, all steps in the evolution from initial requirements to final system are explicitly modelled. Models at various stages of the process are each expressed in the most appropriate formalism. Transformations themselves are also modelled explicitly, so no knowledge is left implicit. Initially, the system is modelled in a formalism amenable to formal analysis and verification (covering all possible behaviours). Subsequently, simulation of the model is performed. The output of this simulation is processed by a checker, which checks it against a set of rules (derived from the requirements). An error found during this checking indicates an error in the design. Note that as even a large number of simulation runs may not cover all possible behaviours of the system, no positive statements about correctness of the model may be made. In the next phase, performance analysis is done to tune the model structure and parameters to satisfy performance requirements. Finally, code is synthesized from the model (if necessary), thus providing a continuous, traceable path from analysis model to deployed system. With appropriate model compilers, the simulation knowledge of the designer is limited to knowledge of suited formalisms (such as DEVS).

3.1 The Pump Control System Case Study

The system used to demonstrate our approach is a Pump Control System (PCS). The PCS has often been used in the real-time systems literature. For example, Burns and Lister used the PCS as a case study to discuss the TARDIS project [14]. We adopt the Pump Control System problem from [14], and with some abstractions, define it as our case study for modelling and simulation based design of a dependable system.

The basic task of the system is to pump to the surface the water that accumulates at the bottom of a mine shaft. The pump must be switched on when the *water-sensor* detects that the water has reached a *high-level* depth, and must be switched off when it detects that the level has been sufficiently reduced (*low-level*). In addition, the pump functionality depends on some atmospheric readings. A *methane-sensor* measures the level of methane in the environment: high levels may cause fire in the shaft if the pump is in operation. A *carbon monoxide-sensor* and an *air-flow sensor* also monitor the environment for critical readings (high for carbon monoxide and low for air-flow) which cause immediate evacuation of the shaft. Critical readings produced by all atmospheric sensors are sent to a human operator, but only critical methane readings cause the pump to switch off. To summarize, the pump is switched **ON** *if the water-level is high and methane-level is not critical, and is switched* **OFF** *if the water-level is low and pump is on; or if the pump is on and methane-level is critical.* The proposed architectural system structure for the PCS is illustrated in Fig. 2.

As all complex and critical applications, the PCS involves some important constraints, namely those of dependability, timing and security. This case study focuses on the dependability requirements defined for the PCS in [14] which dictate that the system is reliable and safe.

Reliability of the pump system is measured by the number of shifts that are lost if the pump does not operate when it should. In order to be considered reliable, our PCS should lose at most 1 shift in 1000.

Safety of the system is related to the probability that an explosion occurs as a result of the pump operating despite critical methane levels. In order to be considered safe, the probability of a possible explosion in our PCS should be less than 10^{-7} during the lifetime of the system.

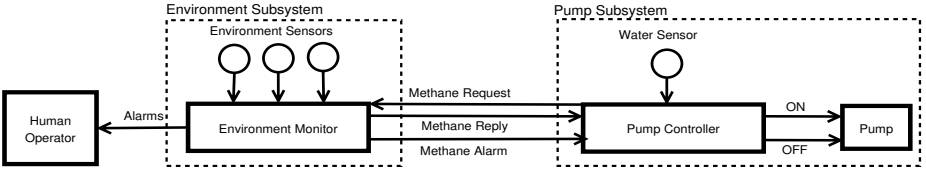

Fig. 2. The Pump Control System Logical Structure.

3.2 Why Use DEVS for the PCS?

The successful development of large-scale complex real-time systems commonly relies on system-theoretic modelling approaches, such as DEVS, or object-oriented approaches such as UML Real-Time. UML-RT is an extension to UML which, in addition to offering constructs to model relationships among components, incorporates the Real-Time Object-Oriented Modelling constructs and is used to model the structural and behavioural aspects of systems. The behaviour of the system is specified in StateCharts by the sequence of signal communication [15]. Contrary to DEVS, in StateCharts we cannot formally specify explicit timing in the specification of models. StateCharts are also

based on multi-component specification and broadcast communication, and the lack of a complete formal definition of UML-RT StateChart semantics hinders the formal specification of structural information. Furthermore, although UML-RT offers important capabilities for modelling real-time systems, it does not provide semantics suitable for simulated time: it prohibits carrying out simulation studies. On the contrary, DEVS separates models from how they may be executed; therefore simulators can be independently developed and verified, thus increasing reusability, formal analysis, and model validation. In addition, DEVS allows the specification of both the structural and behavioural aspects of a system.

The PCS is a reactive discrete-event system: the system's state changes in reaction to external events, such as critical environmental readings. In addition, the PCS is composed of many different interacting subsystems. DEVS, being highly modularized and defining hierarchical coupling of modules, allows for the separation of concerns and a clean model of such a complex system. Since the aim of our approach is to improve the design of a real-time system, we can use the powerful simulation capabilities of DEVS to observe the faulty behaviour in the original PCS model and to predict the system's behaviour under different fault tolerance techniques. From the simulations one can gather statistical data on whether or not dependability requirements are met within the PCS, and evaluate alternative system designs. The above mentioned reasons make DEVS an appropriate modelling formalism for the Pump Control System.

4 Modelling the PCS

4.1 Building the DEVS Model of the PCS

Each subsystem illustrated in Fig. 2 (pump, environment, communication) is modelled as an atomic DEVS whose structure and behaviour encodes the functional requirements of the PCS (Fig. 3). Below is the general description of the system's model.

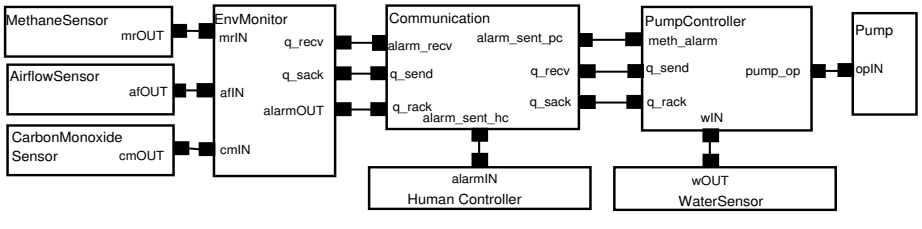

Fig. 3. The Pump Control System Modelled with the DEVS Formalism.

Methane Sensor, Carbon Monoxide Sensor, Airflow Sensor

States: Sensor may either be 'READING' the level of gas or flow in the environment or 'IDLE' between readings.

Output: Upon transitioning from 'READING' to 'IDLE', the sensor outputs the level of gas or flow in the environment at that time.

Environment monitor

States: The monitor may either be processing sensor readings ('PROCESSING'), responding to a query ('QUERYING') or doing nothing ('IDLE').

Output: Upon receiving a query from the Pump Controller through the Communication channel, the monitor responds by sending an acknowledgement which contains a message stating the criticality of the methane level. Upon receiving critical readings from the environment sensors, it outputs alarms. All messages to and from the pump controller or to the human controller are sent through the Communication DEVS.

Communication

States: The communication channel may either be sending alarms ('SEND-ALARM'), sending a query to the environment monitor ('SEND-QUERY') or sending a query acknowledgement to the pump controller ('SEND-ACK'). When it completes either of these tasks, its state is 'IDLE'.

Output: Upon receiving a query from the Pump Controller, it forwards this query to the environment monitor, and once it receives the reply from the environment monitor, it propagates it to the pump controller. When it receives critical alarms, it delivers them to the human and pump controllers.

Pump Controller

States: It may either be processing a water sensor reading and sending an operation to the pump ('PROCESSING-WATER'), processing a methane alarm ('PROCESSING-ALARM'), processing a query acknowledgement ('PROCESSING-ACK'), or doing nothing ('IDLE').

Output: Upon receiving a low-water reading, the pump controller sends an "off" message to the pump to switch it off. If the controller receives a high-water reading, it turns the pump to ready mode and sends a query to the environment monitor: the controller only turns the pump on if the methane level is not critical. If an acknowledgement is received stating that the methane level is high, then the controller turns the pump off, otherwise, it turns it on. Similarly, when the controller receives a methane alarm, it turns the pump off.

Water Sensor

States: It randomly switches between the 'HIGH' and 'LOW' states.

Output: Upon switching, the sensor outputs the state to which it is transitioning.

Human Controller

This is a passive DEVS: it does not react to any input messages and remains constantly 'IDLE'.

4.2 Modelling of Fault Injection in the PCS

As dependability constraints need to be met in addition to functional requirements, a quantitative analysis method for assessing the dependability of the system must also be modelled. For this purpose, many methods have been defined, such as reliability block diagrams, analysis of non-deterministic state graph models, and fault simulation [10]. The latter is a universal approach combining techniques which assume a model of the system, a set of external input/output sequences applied to it, and the possibility to inject faults into it. Most of these techniques can be classified as fault injection techniques,

which consist in adding faults to a system in order to analyze the behaviour. These faults make the system evolve towards different states which are recorded in order to assess the dependability constraints.

Therefore, in addition to modelling the PCS, a model for fault injection must be built. A fault injector could be described as an atomic entity on its own in the coupled DEVS model. However, modelling faults within a specific subsystem itself more accurately represents its real-world faulty behaviour. Our approach consists in provoking a sensor break-down on a periodic basis to simulate a fault which makes the Pump Control subsystem fail. For example, a fault in the methane sensor would generate faulty (noisy) methane readings of the environment, which would be propagated to the environment monitor, and through the communication subsystem to the pump controller. This wrong methane reading could possibly force the pump to shut off when it is not supposed to, or it might fail to cause a critical alarm to be raised. The simulation results should show how the performance varies over time in the absence and presence of faults.

We concentrate here on the consequence of the methane sensor failure on the safety and reliability requirements of the PCS (Section 5.1). To model faulty behaviour of a methane sensor s, we assign to it a probability p of failure. We assume Byzantine failures, i.e. upon failing, sensors produce an erroneous result rather than no result at all. Therefore, s fails by providing erroneous readings with probability p. In practice, a sensor has a very low failure probability, however in this case study, the simulated probability p is chosen to be significantly higher to induce more erroneous states and observable failure of the system. For the methane sensor, we assume $p = 0.1$.

5 Modelling the Fault-Tolerant System

5.1 Failure Scenario in the PCS

Burns and Lister [14] describe four failure situations at the environment, communication and pump subsystems level for the PCS that affect the dependability. To illustrate our approach, we consider the situation in which the environment subsystem provides an incorrect methane reading (when asked by the pump subsystem). The case study focuses on the role of the environment subsystem on safety and reliability, thus upper-bounding the measure of dependability of the system by the dependability of the environment subsystem. We assume that no mechanical failures occur in the communication and pump subsystems and that they do not introduce erroneous state.

The environment subsystem fails in a noisy manner, i.e. it generates incorrect/noisy output. Since we only investigate hardware faults, we assume failures originate in the methane sensor: the subsystem provides incorrect methane readings if it receives such incorrect values from the sensor itself. Therefore, we can generalize the failure scenario to that of the methane sensor providing an incorrect methane reading.

Safety of the System. The safety requirement is threatened if the sensor outputs a falsely low methane reading which causes the pump to operate despite critical concentrations in the environment. This introduces a threat of explosion in the mine

shaft. However, if the sensor outputs a false reading whose criticality is in accordance with the accurate reading, i.e. it is critical when the accurate reading is critical, and not critical when the accurate reading is not critical, then the system is still considered to be safe.

Reliability of the System. The reliability requirement is threatened if the sensor outputs a falsely high methane reading which causes the pump to shut down despite non-critical concentrations in the environment. This causes a loss of shift for the pump.

Safety and reliability can be improved by replication of the methane sensors and applying the TMR technique [14]. This method can also be used for the carbon monoxide and airflow sensors.

5.2 Modelling Fault Tolerance for the PCS

We change the PCS model to integrate fault tolerance based on TMR. A coupled DEVS containing three sets of methane sensors and a voter replace the sensor modelled in Fig. 3. In this case, even if one methane sensor fails, the correct reading can still be determined using the output of the other sensors, and a response from the voter is passed on to the environment monitor. This approach can also be applied to the carbon monoxide and airflow sensors. The fault-tolerant environment subsystem is shown in Fig. 4. In our experiment, we use two different types of voters, a *maximum* voter and a *majority* voter. The maximum voter is a PCS-specific voter in which the highest value received from the replicated sensors is considered as accurate. The interest in the highest value resides in the fact that the system must be safe: if the pump is switched on while methane levels are critical, safety is threatened. Thus, the maximum voter is an appropriate choice for this problem. The majority voter is a well-studied voter that given n results selects the value of the majority. In our case, if majority cannot be decided, the voter falls back on the maximum value.

The fault injection in the sensors is modelled similarly to the PCS model. This allows us to compare the behaviour of the two systems and observe how the performance changes.

6 Simulation and Results

6.1 Performance Metrics Modelling

In the previous sections we showed how the PCS and the fault-tolerant PCS are modelled using DEVS. In order to perform dependability analysis, we model the safety and reliability as dependability metrics to be evaluated while the simulation runs. Each simulation keeps track of the total number of methane readings performed (*TotalMethaneReadings*). A reading m_i is associated with a safety conformance index s_i and a reliability conformance index r_i. These indices are equal to 0 if the reading causes a safety-threatening (for s_i) or reliability-threatening (for r_i) fault, and 1 otherwise. Then safety of the system can be determined by $\sum_{i=1}^{n} s_i$/TotalMethaneReadings, and reliability by $\sum_{i=1}^{n} r_i$/TotalMethaneReadings (where n is equal to $TotalMethaneReadings$).

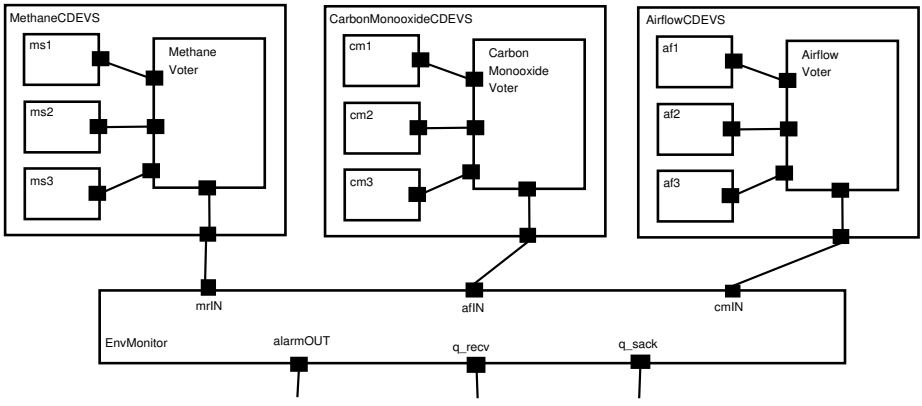

Fig. 4. Fault-tolerant Environment Subsystem of the Pump Control System

6.2 Implementation

Once the system and the constraints are modelled, they are implemented using the PythonDEVS package [16]. This package provides a simulation engine and a class architecture that allows hierarchical DEVS models to be easily defined. Using this framework, each atomic and coupled DEVS described in the model of the PCS, the fault-tolerant PCS using *maximum* voting, and the fault-tolerant PCS using majority voting, can be encoded into a Python class. Python is an interpreted object-oriented programming language, which offers high-level data types and a simple syntax. Its main advantage for the PCS case study is that it is an ideal language for quick and simple application development.

Each Python class representation of a DEVS has four functions defined in it: an internal transition function, an external transition function, an output function and a time-advance function. Next, simulation experiments are set-up to gather statistical data which is representative of the system's behaviour under the specified constraints. The following summarizes the experimental framework:

- **Time advances:** A methane reading is generated every 2s, carbon monoxide every 6s, airflow every 5s, and water level is checked every 10s.
- **Reading Interval:** All environmental readings are integers in the interval $[0, 10]$. We chose integers to avoid the errors common in voters when comparing floating point numbers.
- **Critical Readings:** The critical concentrations are defined in the reading interval to be 7 for methane, 5 for carbon monoxide and 3 for airflow.
- **Simulation Time:** Two sets of experiments are conducted. In the first set, each model is run for a duration of 2000 simulation time units (seconds). This process is repeated 5 times, starting from the same initial state. In the second set, each model is run for a duration of 75000 simulation units to satisfy the law of large numbers. As with the first set, this process is also repeated 5 times. For each of these runs, safety and reliability results are logged and analyzed.

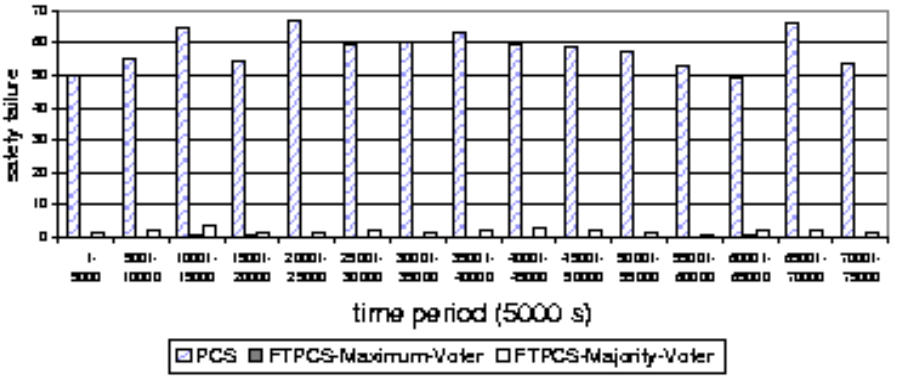

Fig. 5. Safety Results for the Second Set of Simulations.

6.3 Results

Since the results of the first set of simulations are comparable, only results of the second set are analyzed here. These results are an indicator of which voter is best suited for the PCS with regards to system safety and reliability.

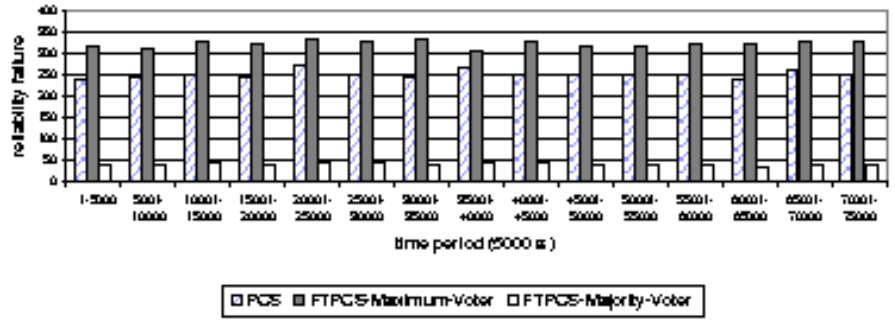

Fig. 6. Reliability Results for the Second Set of Simulations.

Safety. In the initial model, the average failure to satisfy the safety requirement is 2.32%, which is considerably high for a system in which failures are catastrophic in nature. In the fault-tolerant model using the maximum voter, the average safety rises to 99.99% (Fig. 5). It can be concluded that TMR with maximum voting reduces the occurrence of safety-threatening failures. However, there is a notable trade-off between safety and reliability here. This is not surprising as the choice of maximum voter was made to emphasize the safety requirement in such a critical system.

Reliability. In the initial model, the average failure to satisfy the reliability requirement is 10.09%, which is proportional to the probability that was associated with the

methane sensor DEVS of 10% failure. In the implementation with the maximum voter, the reliability percentage falls even lower (Fig. 6). This is explained by the fact that the maximum voter always picks the highest value to output, be it accurate or false. For example, a case where the actual reading is 2, but the false reading received is 8, then 8 is voted to be the correct reading. This approach advocates safety of the system at the cost of reduced reliability of the sensors. In order to attain a fair balance between the safety and reliability requirements, the use of a majority voter is advised. The majority voter implementation results in an average reliability of 98.3%, but a slight decrease in the safety can be seen in Fig. 5. However, this is clearly a solid improvement on the original model and on the maximum voter, while still preserving safety.

6.4 Validation of Results

Over the years, a lot of work has been done on estimating software reliability based on probabilistic models. To compare our simulation-based approach to an analytic one, we perform a probabilistic assessment of the reliability based on the fault-tolerant model that uses majority voting and on the same assumptions as those used for our simulation. We assume that a methane sensor produces an integer reading $r \in [0, 10]$. The sensor either works correctly, or fails with a probability p by outputting a random reading uniformly distributed between 0 and 10.

As discussed previously, reliability fails when a falsely critical reading is sent to the environment monitor although the actual reading is non-critical. There are three cases that lead to a wrong decision by the voter, and can be considered separately. The total probability of the voter failing to decide on the correct output is then equal to the probability that the correct reading is non-critical (which is 7/11) multiplied by the sum of the probabilities corresponding to the cases listed below:

- one sensor outputs a correct reading, two sensors output equal, critical and false reading: $3 * (1-p) * (p * 4/11) * (p * 1/11)$
- all three sensors output wrong readings, but at least two are equal, critical and false reading: $p^3 * ((4/11)(1/11) + 2(7/11)(4/11)(1/11))$
- all three sensors output wrong distinct readings, and at least one is critical: $p^3 * (1 - 7/11 * 6/11 * 5/11) * (10/11 * 9/11)$

Since we assume that $p = 0.1$ for the methane sensor, this leads to a majority voter failure probability of 0.0061, or a reliability of 99.39%. The results of our simulation indicated a reliability of 98.3%, clearly comparable to the results derived from the analytic model.

This probabilistic assessment leads to exact and precise results, but in cases where the problem is non-linear, the equations may become very complex and impossible to solve. On the other hand, the approach presented in this paper is especially effective for complex systems for which deriving mathematical models is not feasible. One might argue that this approach requires extensive work in designing and encoding the models, and in analyzing the simulation results. However, models are easily derived from the requirements and logical structure of the system. Furthermore, the choice of modelling

formalism and programming language make for a modular implementation, and if tools are available which automatically generate the applications, the process can be speedy. Lastly, simulation results are simple to analyze as they are derived from such simple equations as those described in Section 6.1. Mathematical models do not have these advantages. However, probabilistic models can be useful as a validation method for modelling and simulation based approaches as well as provide solutions to rare-event cases.

7 Conclusion

In most complex systems today, it is crucial to guarantee that the dependability requirements are successfully achieved. Methods should be provided which can accurately assess what level of dependability has been attained by a system. In this paper, we have presented a modelling and simulation-based development process targeted towards dependable systems, and have demonstrated it through an application to the safety-critical Pump Control System.

A continuity was maintained throughout the development process. We started from requirements, mapped these to a DEVS model, extended the model to consider the dependability constraints, defined performance metrics, implemented the model using the PythonDEVS framework, and performed simulations whose results reflected the safety and reliability of the system. DEVS is deemed the most appropriate formalism for modelling both the system under study and the fault tolerance techniques. This, as discrete-event models are clearly at the right abstraction level, and because of the compositionality of the DEVS formalism. Fault tolerance, more specifically TMR, was used as a means to achieve dependability. In this approach, two types of voters were used, and the simulation results were inspected to decide which voter best satisfied the dependability requirement. The results indicated that this outlined method improved the dependability levels of the example system.

We have shown how models can be useful for designing dependable systems: a model can be extended to address possible failures and to incorporate fault tolerance techniques that overcome them. This approach allows us to predict behaviour and estimate system dependability, and it enables an informed decision on which fault tolerance technique to apply. If such a step is taken during the analysis and design phase of any project, development cost is reduced as an optimal system is built right the first time, while fault tolerance is addressed earlier on in the development cycle, and simulation results emulate the expected behaviour of the dependable system.

We plan to further investigate a generic *process* for the analysis and design of dependable systems. Furthermore, we will use the fault-tolerant models to synthesize the final application.

References

[1] Gray, J., Rossi, M., Tolvanen, J.P., eds.: Domain-Specific Modeling with Visual Languages. Volume 15 of Journal of Visual Languages & Computing. Elsevier Science Publishers (2004)

[2] Vangheluwe, H., de Lara, J.: Domain-specific modelling for analysis and design of traffic networks. In Ingalls, R., Rossetti, M., Smith, J., Peters, B., eds.: Winter Simulation Conference, IEEE Computer Society (2004)
[3] Pfeifer, H., von Henke, F.W.: Formal modelling and analysis of fault tolerance properties in the time-triggered architecture. In: 5th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems. (2004)
[4] Boue, J., Arlat, J., Crouzet, Y., Petillon, P.: Verification of fault tolerance by means of fault injection into VHDL simulation models. In: Contrat Esprit DeVa Project. (1996)
[5] Zeigler, B.P.: Multifacetted Modelling and Discrete Event Simulation. Academic Press (1984)
[6] Zeigler, B.P., Praehofer, H., Kim, T.G.: Theory of Modeling and Simulation, Second Edition. Integrating Discrete Event and Continuous Complex Dynamic Systems. Academic Press (2000)
[7] Cheon, S., Seo, C., Park, S., Zeigler, B.: Design and implementation of distributed DEVS simulation in a peer to peer network system. In: 2004 Advanced Simulation Technologies Conference, Design, Analysis, and Simulation of Distributed Systems Symposium 2004 (2004)
[8] Chi, S., Lee, J.: DEVS-based modeling and simulation for intelligent transportation systems. In Sarjoughian, H.S., Cellier, F.E., eds.: Discrete event modeling and simulation: A tapestry of systems and AI-based theories and methodologies. Springer-Verlag (2001) 215–227
[9] Filippi, J., Chiari, F., Bisgambiglia, P.: Using jDEVS for the modeling and simulation of natural complex systems. In: SCS AIS 2002 Conference on Simulation in Industry. Volume 1. (2002)
[10] Geffroy, J.C., Motet, G.: Design of Dependable Computing Systems. Kluwer Academic Publishers (2002)
[11] von Neumann, J.: Probabilistic logics and the synthesis of reliable organisms from unreliable components. In Shannon, C.E., McCarthy, J., eds.: Annals of Math Studies. Princeton University Press (1956) 43–98
[12] Laprie, J.C.: Dependable computing and fault tolerance : Concepts and terminology. In Meyer, J.F., Morgan, D.E., eds.: 15th FTCS. (1985)
[13] Lee, P.A., Anderson, T.: Fault tolerance - principles and practice. In: Dependable Computing and Fault-Tolerant Systems. 2nd edn. Springer Verlag (1990)
[14] Burns, A., Lister, A.: An architectural framework for timely and reliable distributed information systems (TARDIS): Description and case study. Technical report, University of York (1990)
[15] Huang, D., Sarjoughian, H.: Software and simulation modeling for real-time software-intensive system. In: Proceedings of the 8th IEEE International Symposium on DS-RT. (2004)
[16] Bolduc, J.S., Vangheluwe, H.L.: The modelling and simulation package pythonDEVS for classical hierarchical DEVS. Technical report, McGill University (2001)