Automated Safety Analysis for Domain-Specific Languages

Richard F. Paige, Louis M. Rose, Xiaocheng Ge, Dimitrios S. Kolovos, and Phillip J. Brooke

Department of Computer Science, University of York
{paige, louis, xchge, dkolovos}@cs.york.ac.uk
School of Computing, University of Teesside pjb@scm.tees.ac.uk

Abstract. Critical systems must be shown to be acceptably safe and secure to deploy and use in their environment. But the size, scale, heterogeneity, and distributed nature of these increasingly complex systems makes them difficult to verify and analyse. Additionally, domain experts use a variety of languages to model and build their systems. We present an automated safety analysis technique, Fault Propagation and Transformation Analysis, and explain how it can be used for automatically calculating the *failure behaviour* of an entire system from the failure behaviours of its components. We outline an implementation of the technique in the Epsilon model management platform, thus allowing it to be used in combination with state-of-the-art model management languages and tools, and making it applicable to a variety of different domain-specific modelling languages.

1 Introduction

Complex systems exhibit emergent properties as a result of composing heterogeneous components. These components may be distributed, and may also have substantial performance, timing, safety, and security requirements. The scale and complexity of these systems make it difficult to apply general-purpose verification and validation technology – such as model checkers, simulators, and theorem provers – to obtain the guarantees of acceptable behaviour that are required. Obtaining guarantees is particularly important for safety critical systems, which normally must be certified as acceptably safe according to relevant standards, before they are deployed in the field.

Safety analysis for complex systems is an open field of research. For high-integrity real-time systems (HIRTS), *automated* safety analysis can help to achieve the substantial requirements for reliability and safety necessary for these kinds of systems to achieve certification. Safety analysis techniques are novel when contrasted with traditional software analyses, which tend to emphasise determining a product's correctness – e.g., through proof, model checking, simulation, or abstract interpretation. For HIRTS, it is of critical interest to know how a system behaves in the presence of failure, regardless of whether that failure is in the environment, or due to internal software or hardware error. Given

an understanding of a system's behaviour in the presence of failure, methods to mitigate potential hazards can be determined and engineered.

Manual safety analysis is notoriously expensive for all systems; anything that can be done to help to automate the process of understanding system behaviour in the presence of failures will be of benefit to industry. Moreover, safety analysis is not in general compositional – even small changes to components (and their corresponding failure behaviour) generally means that the whole safety analysis has to be performed again for the entire system. Finally, engineers of safety critical systems often use a variety of domain-specific languages (DSLs), including profiles of UML, Matlab, Simulink, Stateflow, AADL, SysML, MASCOT [13], et cetera; safety analysis that is applicable to all these domain-specific languages (and others) will be of substantial value.

This paper presents a fully automated and compositional safety analysis technique applicable to domain-specific languages. The technique, *fault propagation and transformation analysis*, is outlined and an implementation in the Epsilon model management toolset [10], is described in detail. The value of having an implementation in a state-of-the-art and standards compliant toolset like Epsilon, built atop of Eclipse, is also explained, particularly for supporting analysis on domain-specific and heterogeneous models.

We start with a brief overview of previous work and background, discuss the notion of failure modelling for components, and how it supports the compositional reasoning essential for safety analysis of complex systems. We describe an implementation to allow the safety analysis to be applicable to models represented using different modelling technologies (such as EMF and MDR), and discuss how to customise the analysis for different domain-specific languages.

2 Background and related work

2.1 Components and failures

Failure and safety analysis is generally applied to component or architectural models of safety critical systems. In these models (which may be represented using one of a number of different DSLs), a component is a building block of a system, and may be represented in hardware or software. A component has input ports and/or output ports, and transfers inputs to outputs. Components may exhibit expected behaviour (e.g., according to a specification such as a pre- and postcondition), but may also exhibit *failures*. A failure is any behaviour of a component or system which deviates from specified behaviour [5]. Failures arise, can be propagated, and can also be transformed in a system, e.g., as a result of an accident or incorrect implementation. In order to determine the failure behaviour of a system, it is necessary to be able to understand, and model, the failure behaviours of the system's components.

There has been previous work on modelling and understanding the failure behaviour of systems. Traditional safety engineering techniques include Failure Modes and Effects Analysis (which is a manual process) [7], HAZOPs guidewords-based analyses, Fault Tree Analysis, and finite state analysis [6]. Fenelon et al

introduced FPTN [4], a notation for explicitly representing the failures behaviour of components, and integrated FPTN with a typical safety engineering process. However, FPTN possessed no tool support. Overall, few, if any, of these approaches have been integrated with Model-Driven Engineering standards and tools. An exception is work on integrating Fault Tree Analysis with UML, e.g., as carried out by Jürjens et al; this integration was specifically for UML, and did not support automated compositional reasoning about failures, rather focusing on providing tool support for building fault trees and calculating probabilities of faults occurring..

Wallace [14] proposed using a model of HIRTS system architecture as the basis for safety analysis. The failure behaviours of the components are determined and modelled when analysing the system. The connections between units are communication protocols. Because a communication protocol also has its own potential failure behaviour, the protocols in the model must be treated identically to the computational components of the system – i.e., their failures are also modelled.

In [14], a component can introduce failures (e.g., because of an exception or crash), or may propagate failures (e.g., data that is erroneous when it arrives at a component remains erroneous when it leaves the component), or transforms a failure into a different kind of failure (e.g., data that arrives late may thereafter arrive early). Furthermore, a component may correct or mask failures that it receives. Thus, when a component receives as input a particular kind of failure, it generates one of the following responses.

$$output = \begin{cases} normal \\ same\ failure \\ different\ failure \end{cases}$$

To support automated failure analysis, we must be able to connect models of component failure behaviour to a system model. This can be done by representing the behaviour of architectural components – such as hardware, wires, and network connections – using failure models. In our implementation in Epsilon, we do this by effecting a model transformation (though it is not a traditional *mapping* transformation in the classification of Czarnecki [2]).

Conceptually, failure analysis can be applied to any model, whether it is represented in UML 2.x, SysML, AADL, or another domain-specific language.

2.2 Automating failure analysis

Assume that we have a component-and-connector model of a system, e.g., in a DSL. The components in the system can be individually analysed – in isolation, from the rest of the system – for their failure behaviour in response to potential failure stimuli. This behaviour should be determined by domain and safety experts, and should consider all possible failures on input. The analysis can follow the conventional HAZOP/SHARD [12] identification of *types* of failures through

a set of *guidewords*, such as: value failures (e.g., data is stale); timing failures (e.g., data is arriving later); and sequence failures (e.g., omission).

Following [14], we capture the failure responses of a component to its input in a simple pattern-based modelling language. Using `*` to indicate normal (no failure) behaviour, the following four expressions denote example source, sink, propagation, and transformation behaviours for a trivial single-input single-output component (the generalisation to multiple inputs and multiple outputs is straightforward, and is illustrated in Section 4).

$$\begin{aligned} * &\rightarrow \text{late} && \textit{(failure source)} \\ \text{early} &\rightarrow * && \textit{(failure sink)} \\ \text{omission} &\rightarrow \text{omission} && \textit{(failure propagation)} \\ \text{late} &\rightarrow \text{value} && \textit{(failure transformation)} \end{aligned}$$

The first line says that any input leads to late output, whereas the second says that early input (data arriving before its time) leads to no error, i.e., the component sinks all errors. The third line says that an omission failure is propagated by the component. The most complicated failure behaviour is generally transformational – i.e., the last line above, where a late input leads to a value error — the wrong value being output.

A typical component will have its failure behaviour modelled by a number of patterns of this form, and the cumulative effect is its overall *behaviour*. [14] calls this the *FPTC behaviour* of a component.

To represent the system as a whole, every element of the architectural model – both components and connectors – is assigned FPTC behaviour. Given this, we can automatically calculate the failure behaviour of a whole system as follows (see [14] for a formal definition). Each model element that represents a relationship is annotated with sets of *tokens* (e.g., late, early, value), which represent all possible failures that can be propagated by this dependency. In other words, we are informally treating the architectural model as a token-passing network. As a result of this annotation, we can calculate the failure behaviour of the system by calculating the *maximal* token sets on all dependencies in the model. This turns out to be a fixpoint calculation (presented formally in [14]). Informally, the calculation works as follows. Starting with the singleton set containing the *no failure* (`*`) token as a label on every dependency, the FPTC behaviour at every component model element is 'run', using the token sets on input dependencies as the inputs to the FPTC behaviours. The output failure tokens of each component are accumulated on the outgoing dependencies, and the system continues to run until a fixed point is reached, i.e., the token sets no longer change.

The calculation must terminate, because the set of failure types must be finite. [14] also shows that the calculation produces the same result no matter in what order the relationships are analysed.

3 Implementation in Epsilon

We have implemented the failure analysis in the Epsilon model management platform, under Eclipse. By using Eclipse, we can exploit its mechanisms for

metamodelling, modelling, and extension, as well as its substantial tool support via plug-ins. In particular, we can manipulate the models used as the basis for the failure analysis using other Eclipse tools, particularly model transformation tools (e.g., to transform the architectural models into representations that can be imported by other tools), model merging tools, and simulation tools.

The implementation has been constructed atop the existing Epsilon[1] development tools for Eclipse. Epsilon provides a *model management* framework, via a suite of integrated languages. Epsilon provides a base language – the Epsilon Object Language (EOL) [11] – which supports basic model manipulation, e.g., traversal of models, querying models, modifying models. EOL has many similarities to the OMG-standard Object Constraint Language (OCL), but is fully executable and metamodel-independent; thus, EOL (and Epsilon) can be used to manage models from any language. By implementing the failure analysis approach within Epsilon, we thus immediately obtain independence from UML-based languages, but also technology independence, because EOL can be used to manage models from different technologies such as EMF, MDR/MOF, Z models, and XML. This is particularly critical for complex systems which exhibit heterogeneity.

An example of an EOL specification is in Listing 1.1. It allows easy traversal of models, and modification of models, without having to operate directly at the level of XML/XMI. The example below demonstrates the use of EOL for comparing two models: a UML model and a Database model. The example checks that for each class in the UML model, there is a table in the Database model with the same name; an indicative message is produced for each UML class.

Listing 1.1. EOL Program

```
1 for (class in UML!Class.allInstances()){
2   if(DBMS!Table.allinstances().exists(t|t.name=class.name)){
3     ('Found matching table for class '+class.name).println();
4   }
5   else {
6     ('No matching table for '+class.name).println();
7   }
8 }
```

The Epsilon platform includes other model management languages that have been built on, and thus inherit from, EOL. These include a model-to-model transformation language, a model merging language, a model-to-text transformation language, a validation language, and a refactoring language, amongst others. Further details can be found in [10].

The FPTC analysis has been implemented and encoded directly in EOL, based on a lightweight and reasonably generic metamodel for architectural modelling; the metamodel is shown in Fig. 1. EOL was used because the model management task to be completed – calculating a fixpoint on a model – is iterative, and EOL is the only language in the platform providing iterative constructs.

[1] www.eclipse.org/gmt/epsilon

The metamodel that we use as the basis of the FPTC calculations is intended to be generic so that it can (a) provide sufficient infrastructure for the FPTC calculations; and (b) make it reasonably straightforward to use as the target of model-to-model transformations from other architectural modelling DSLs, such as UML 2.x, SysML, and AADL. We have implemented several simple transformations for such source languages.

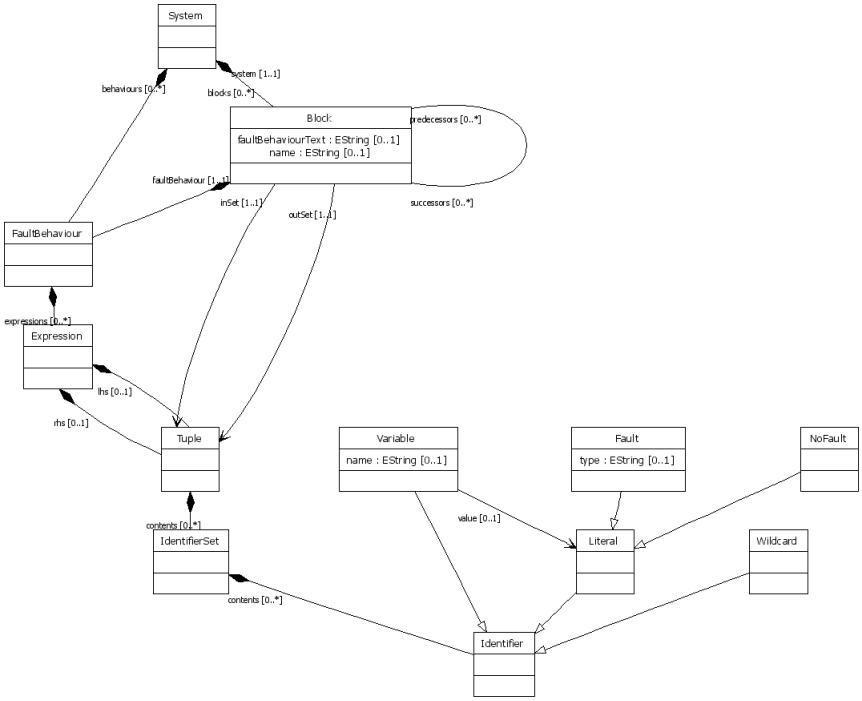

Fig. 1. Example metamodel for architectural modelling

In the architectural language of Fig. 1, systems are made up of blocks (which represent both components and connectors). The system overall, and individual blocks, have fault behaviour, represented as expressions. Expressions are made up of a number of tuples (which correspond to the patterns we discussed earlier). These tuples include sets of identifiers, where an identifier can be a wildcard (i.e., no-fault behaviour), a literal (i.e., a domain-specific kind of fault), or a variable.

The actual implementation, written in EOL, encodes the algorithms described earlier. A certain amount of simple EOL infrastructure needs to be provided (e.g., to record the behaviour of blocks, variables, and literals, and to reset these behaviours across different runs of the analysis). The remainder of the implementation is more complex, and can be subdivided into four main

parts (not including any visual representation of the output of the analysis, nor how individual FPTC behaviours are expressed – we discuss this afterwards):

- the pattern matching, e.g., to match failure behaviours with inputs; this is written as a *model comparison* operation in EOL. We could equally do this in the Epsilon Comparison Language (ECL) but because the pattern matching that we need to do is straightforward and not rule-based – and we need to use the results of the matching in further EOL programs – we encode the comparison directly in EOL. This is one of the benefits of having an executable base language in Epsilon that is also computationally complete.
- the propagation behaviour, i.e., what happens when a component or connector propagates failure behaviour to its environment. As this is an algorithmic calculation, we implement this with EOL.
- the transformation behaviour, i.e., what happens when a component or connector generates new failure behaviour to its environment, based on specific input behaviour. This could be implemented using the Epsilon Transformation Language (ETL) [8] or EOL. We chose the latter for reasons similar to the model comparison phase: the transformation we need to carry out is not a mapping, and is predominantly algorithmic instead of rule-based. As such, EOL was a better fit for this transformation problem versus ETL.
- the overall system analysis, which is a fixpoint calculation over the system model.

The matching behaviour in EOL is described in Listing 1.2. This implements a pattern matching on blocks and sets of identifiers. The pattern matching is implemented as a set of overloaded operations, called *matches*; one *matches* operation is defined for each type of model element that can be matched, e.g., blocks, faults, variables, identifier sets, etc. Effectively, each operation simple compares an input (consisting of failure behaviour) against the behaviour of a model element and returns true or false. We show three examples: for matching sets of identifiers, for matching faults, and for matching no-fault behaviour; other match operations are direct transliterations of the ones we show.

Listing 1.2. EOL Pattern Matching

```
1  operation IdentifierSet matches(inSet : IdentifierSet) : Boolean {
2
3   for (identifier in self.contents) {
4    for (inSetIdentifier in inSet.contents) {
5     if (identifier.matches(inSetIdentifier)) {
6      return true;
7     }
8    }
9   }
10  return false;
11 }
12
13 operation NoFault matches(identifier : Identifier) : Boolean {
14  if (identifier.isTypeOf(NoFault)) {
```

```
    return true;
  } else {
    return false;
  }
}

operation Fault matches(identifier : Identifier) : Boolean {
  if (identifier.isTypeOf(Fault)) {
    return identifier.type = self.type;
  } else {
    return false;
  }
}
```

Failure propagation behaviour is illustrated in Listing 1.3. This EOL program calculates output behaviour of a block from input behaviour. Effectively, when the *propagate* operation is applied to a specific block in an architectural model, it iterates through all successor blocks (i.e., all blocks that it is connected to). After obtaining the input token set for the current block, it simply propagates all input faults to the output block.

Listing 1.3. EOL propagation behaviour

```
operation Block propagate() {
  var index : Integer := 0;

  for (successor in self.successors) {
    var inSet : IdentifierSet;
    var outSet := self.outSet.contents.at(index);

    -- Retrieve the corresponding in-set for this out-set
    var pIndex : Integer := 0;
    for (predecessor in successor.predecessors) {
      if (predecessor = self) {
        inSet := successor.inSet.contents.at(pIndex);
      }
      pIndex := pIndex + 1;
    }

    -- Propagate identifiers from out-set to in-set
    for (identifier in outSet.contents) {
      inSet.contents.add(identifier.clone());
    }
    index := index + 1;
  }
}
```

The transformation behaviour is the most complicated part of the analysis. The EOL program implementing the transformation calculates new failure behaviour from input behaviour. It applies the *matches* operations presented earlier to match input failure behaviours against failure behaviours of the component.

If there is a match on the left-hand side of a pattern, then the right-hand side failure behaviour is generated on the output of the block. The main part of the functionality is in operation *transform*, shown below.

Listing 1.4. EOL transformation behaviour

```
1  operation Block transform() : Boolean {
2   -- Determine which expressions match
3   var applicable : Sequence(Expression);
4   for (exp in self.faultBehaviour.expressions) {
5    if (exp.lhs.matches(self)) {
6     applicable.add(exp);
7    }
8   }
9
10  var result : Boolean := false;
11  var selected : Expression;
12
13  self.toString().println();
14
15  if (applicable.size() > 0) {
16   if (applicable.size() = 1) {
17    selected := applicable.at(0);
18   } else {
19    selected := applicable.mostSpecific();
20   }
21
22   selected.toString().println();
23   result := selected.applyTo(self);
24   self.toString().println();
25  }
26  ''.println();
27
28  return result;
29 }
```

There are a few subtleties to implementing the transformation behaviour. When matching input failures against failure behaviours of a component, there may be several matches; this is recorded in the EOL program via the variable *applicable*. This is an artefact of allowing wild-card specifications of behaviour (i.e., any fault is matched). To deal with this issue, we always select the most specific match; this is implemented in an EOL operation called *mostSpecific()*.

The second subtlety is in copying failure values to the output of a block. A component may have several failures on its outputs (and indeed, it may have many outputs), and we must be careful to record all of them in the output expressions for the block. This is handled in the EOL operation applyTo(), which applies a model of failure behaviour to a block's inputs. While we omit the details of applyTo(), it is a good example of a transformation that is not inherently a mapping, and which would be more concise expressed using an algorithmic specification. Another example of such a transformation was presented by Conmy

[1], where the transformation was intended to generate large numbers of stable configurations of an adaptive system. These transformations are similar because both involve iterative processing of models, rather than rule-based processing. Conmy contrasted mapping transformations against algorithmic transformations in [1] in more detail.

Finally, the overall system analysis is encoded in Listing 1.5. The failure analysis is launched on the full system by the EOL run-time. The analysis first initialises all blocks with their failure behaviour, and then calculates the output sets on all blocks, until no output set changes, i.e., a fixpoint has been reached.

Listing 1.5. EOL failure analysis

```
1  System.allInstances().at(0).doFailureAnalysis();
2
3  operation System doFailureAnalysis() {
4    -- Initialise
5    for (block in Block.allInstances()) {
6      block.initialise();
7    }
8
9    var blocksChanged : Boolean := true;
10
11   while (blocksChanged) {
12     blocksChanged := false;
13
14     -- Calculate out sets
15     for (block in Block.allInstances()) {
16       blocksChanged := block.transform() or blocksChanged;
17     }
18
19     -- Calculate new in sets
20     for (block in Block.allInstances()) {
21       block.resetInSet();
22     }
23     for (block in Block.allInstances()) {
24       block.propagate();
25     }
26
27     ('================================================').println();
28   }
29 }
```

Epsilon is integrated with Eclipse GMF, so it is possible to create customised GMF editors and visualisations of the results of the FPTC analysis, and of the architectural models themselves. We present an example of this in the next section.

This implementation has the advantage of support abstract specification, executability, and cross-platform capabilities: the failure analysis will then be applicable to any model that can be encoded in Eclipse's EMF format. As many commercial and open-source tools already support EMF (or provide injection

to EMF/Ecore) this provides substantial support for architectural modelling frameworks.

4 Example

In this section, we introduce a concrete example to illustrate the functionality provided by our FPTC implementation. Consider the architectural model shown in Figure 2. The model is written in a DSL for real-time systems. The depicted system comprises four software components, connected using three instances of a signalling communication protocol. This protocol uses a destructive (non-blocking) write, and a destructive (blocking) read.

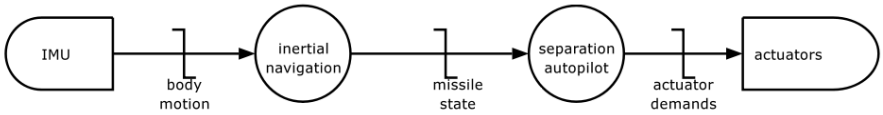

Fig. 2. Architectural model of the exemplar system.

We have used a simple GMF editor for creating this model. The model must now be transformed to include failure behaviour, so that we can perform the failure analysis. FPTC behaviours are most easily expressed in a textual format. In order to support this, we have integrated Epsilon with a model-generating parser specified using oAW's xText [3]. This allows the FPTC behaviour to be easily expressed in a format similar to what was presented earlier. The FPTC behaviours of the individual components and connectors is listed in Table 1. These behaviours have been determined by domain experts knowledgeable about the individual components and connectors and their properties.

These experts have determined that the inertial navigation and separation autopilot components both propagate any faults that they receive. In addition the separation autopilot component acts as a source for *stale_value* and *detectable_value* faults. The signalling communication protocol exhibits a rather more complex failure behaviour, and comprises three non-trivial expressions. The first states that, as the protocol utilises a blocking read, should the supplier provide a value earlier than the receiver expects, no fault is produced. In the case where the communications protocol fails to relay a message (an omission), the receiver may block indefinitely, causing it to be delayed (encoded as a late fault). When the communications protocol duplicates a message sent from the supplier (a commission), the receiver may proceed with an incorrect value. Additionally, the protocol simply propagates all other categories of fault.

Having used our GMF-based editor to record the results of our behavioural analysis of the individual components, we can inject various different types of faults to potential sources of errors, and run the FPTC analysis to determine how the system would respond to these types of failures. For example, injecting

Component	Behaviour
Inertial Navigation	v → v
Separation Autopilot	* → *stale_value* * → *detectable_value* v → v
Signal Comms Protocol	*early* → * *omission* → *late* *commission* → *value* v → v

Table 1. Behavioural properties of components.

an omission fault on the IMU component and executing our simulation toolchain yields the results depicted in Figure 3. By examining the faults produced by the actuator demands, it can be seen that the actuators may receive faults from the set {*, *stale_value*, *detectable_value*, *late*}.

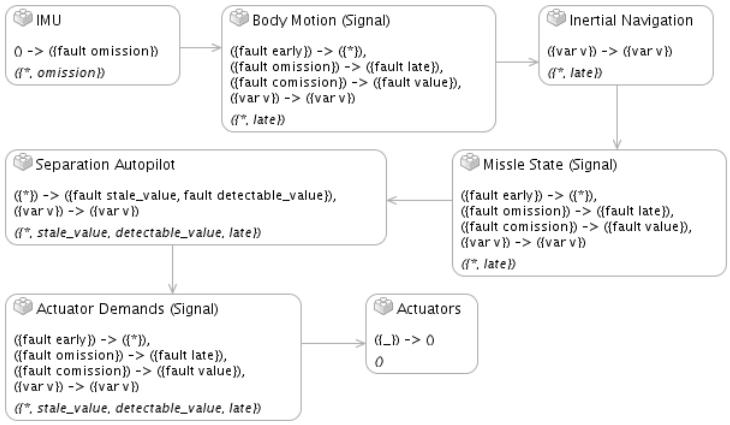

Fig. 3. Results of executing the simulation on the system. The faults produced by each component are shown in italics.

Suppose the design is now changed such that a second IMU is introduced in order to provide two-lane redundancy. Additionally, instead of the signal, a pooling communications protocol is used between the IMU and inertial navigation components. Unlike the signalling protocol, the pooling protocol provides a buffer from which the receiver may non-destructively read. This leads to the rather different failure behaviour shown in Table 2. Should the pooling protocol omit a message from receiver to supplier, the receiver may read a previous (stale) value from the buffer. Similarly, if the supplier is late sending a message to the receiver, the receiver may read a stale value from the buffer. Finally, if

the protocol duplicates a message sent from supplier to receiver, the receiver proceeds as expected, due to the data being buffered by the protocol.

Component	Behaviour
Pool Comms Protocol	$early \rightarrow *$ $omission \rightarrow stale\_value$ $late \rightarrow stale\_value$ $commission \rightarrow *$ v → v

Table 2. Behavioural properties of the components in the simple system.

As can be seen in Figure 4, executing the simulation on the new model highlights that the set of faults propagated to the actuator from the actuator demands is {*, *stale_value*, *detectable_value*}. As such, we can conclude that the new model provides mitigation against faults with the categorisation *late*, whereas the original model does not.

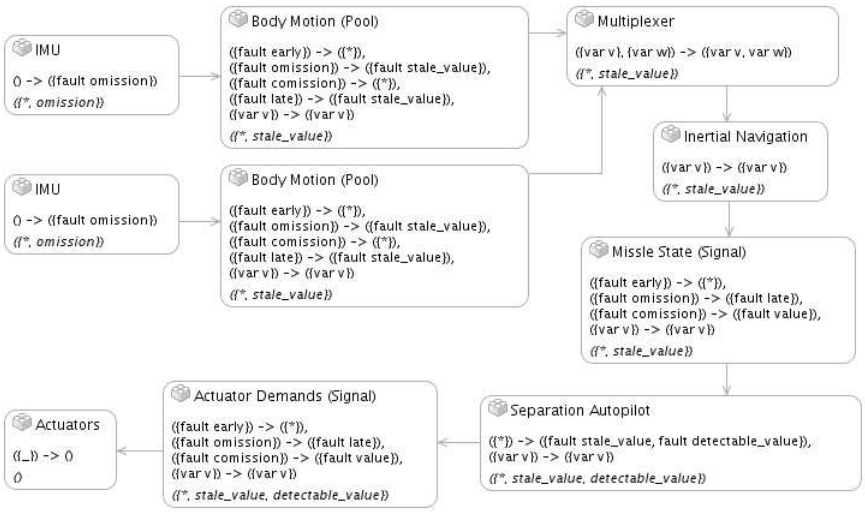

Fig. 4. Results of executing the simulation on the modified system.

This example illustrates the results that can be obtained by applying FPTC, and the lightweight nature of its analysis – we were able to change the architectural model to introduce different failure, and re-run the analysis to calculate the overall effect on the system. The ability to automatically and quickly analyse models makes the failure analysis technique particularly valuable for complex and critical systems development.

5 Discussion and Conclusions

There are two additional, technical points regarding the FPTC implementation in Epsilon that are worth noting:

- Initial implementations of the failure analysis did not provide detailed error checking, e.g., to ensure that erroneous or inconsistent failure behaviours were not specified for components. For example, consider a component which was accidentally specified to deliver data both late and early; this should ideally be caught statically, before the FPTC calculation has been run. As part of the Eclipse/EOL implementation, we have exploited the availability of the Epsilon Validation Language (EVL) [9] for specifying well-formedness rules and constraints on the model. This helps us catch errors at an early stage. We point out that this consistency/constraint checking is also fully automated.
- As mentioned, we have provided support for constructing customised graphical interfaces by integrating Eclipse's GMF (Graphical Modelling Framework) with EOL. This allows the results of the failure analysis to be presented in ways suitable and appropriate for exploration by domain experts. This in itself is a novelty and provides functionality that is generally useful, not just for FPTC.

We have applied the FPTC analysis as part of work carried out in the Defense and Aerospace Research Partnership project at the University of York, in collaboration with BAE Systems, Rolls-Royce, QinetiQ, and the Ministry of Defense. The FPTC toolset has been applied to a number of case studies of very different models. The results demonstrated that the analytic technique was (a) scaleable; (b) efficient; and (c) produced insightful results. Indeed, unexpected failure behaviour was detected in at least one case study. These unexpected results were reported to the relevant engineers, who determined that the failure behaviour was indeed accurate, and was in fact mitigated by hardware elsewhere in the system (hardware that was not modelled).

We are currently working on extensions to FPTC to support probabilistic analysis, i.e., where engineers can indicate the probability of particular types of failures occurring. This requires extension not only to the theory underpinning FPTC (specifically, the calculation of output token sets becomes much more complicated, since conditional probabilistic reasoning must be used), but also some extensions to Epsilon in order to efficiently support matrix calculations, which are an appropriate way to implement probabilistic analysis. As well, we are developing transformations for popular architectural modelling languages (such as AADL and SysML) into the analysis metamodel presented in this paper, so that FPTC can be used for these domain-specific languages as well.

Acknowledgements. The work in this paper is partially supported by the European Commission via the MODELPLEX project, co-funded under the "Information Society Technologies" Sixth Framework Programme (2006-2009), and

the Engineering and Physical Sciences Research Council (EPSRC) under the Large-Scale Complex IT Systems project, supported by research grant EP/F001096/1.

References

1. Philippa Conmy and Richard Paige. Challenges when using Model-Driven Architecture in the development of safety critical software. In *Proceedings of 4th Workshop on Model-Based Methodologies for Pervasive and Embedded Software*. IEEE Press, 2007.
2. Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.
3. Sven Efftinge. xText reference document, www.eclipse.org/gmt/oaw, 2007.
4. Peter Fenelon and John A. McDermid. An integrated toolset for software safety analysis. *The Journal of Systems and Software*, 21(3):279–290, June 1993.
5. Lars Grunske. Towards an integration of standard component-based safety evaluation techniques with saveccm. In *QoSA*, pages 199–213, 2006.
6. Constance L. Heitmeyer, James Kirby, Bruce G. Labaw, Myla Archer, and Ramesh Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. Software Eng.*, 24(11):927–948, 1998.
7. IEC. Analysis techniques for system reliability: Procedures for failure mode and effect analysis. International Standard 812. IEC Geneva, 1985.
8. Dimitrios Kolovos, Richard Paige, and Fiona Polack. The epsilon transformation language. In *International Conference on Model Transformation 2008*. LNCS 5063, Springer-Verlag, 2008.
9. Dimitrios Kolovos, Richard Paige, and Fiona Polack. On the evolution of OCL for capturing structural constraints in modelling languages. In *Rigorous Object-Oriented Methods*. Springer, 2008.
10. Dimitrios S. Kolovos and Richard F. Paige. Epsilon model management platform, www.eclipse.org/gmt/epsilon, 2008.
11. Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The Epsilon Object Language (EOL). In *ECMDA-FA*, pages 128–142, 2006.
12. J A McDermid, M Nicholson, D J Pumfrey, and P Fenelon. Experience with the application of HAZOP to computer-based systems. In *Compass '95: 10th Annual Conference on Computer Assurance*, pages 37–48, Gaithersburg, Maryland, 1995. National Institute of Standards and Technology.
13. H R Simpson. The MASCOT method. *Software Engineering Journal*, 1(3):103–120, March 1986.
14. Malcolm Wallace. Modular architectural representation and analysis of fault propagation and transformation. In *FESCA'05*. ENTCS, Elsevier, April 2005.