Probabilistic Failure Propagation and Transformation Analysis

Xiaocheng Ge, Richard F. Paige, and John A. McDermid

Department of Computer Science, University of York, UK
{xchge,paige,jam}@cs.york.ac.uk

Abstract. A key concern in safety engineering is understanding the overall emergent failure behaviour of a system, i.e., behaviour exhibited by the system that is outside its specification of acceptable behaviour. A system can exhibit failure behaviour in many ways, including that from failures of individual or a small number of components. It is important for safety engineers to understand how *system* failure behaviour relates to failures exhibited by individual components. In this paper, we propose a safety analysis technique, *failure propagation and transformation analysis* (FPTA), which automatically and quantitatively analyses failures based on a model of failure logic. The technique integrates previous work on automated failure analysis with probabilistic model checking supported by the PRISM tool. We demonstrate the technique and tool on a small, yet realistic safety-related application.

Keywords: failure, safety analysis, probabilistic analysis, component-based system.

1 Introduction

Modern systems, comprising hardware and software components, are becoming increasingly complex. The design and development of these complex systems is challenging, because engineers need to deal with many functional and non-functional requirements (e.g., safety, availability, and reliability requirements), while keeping development cost low, and the engineering life-cycle as short and manageable as possible.

Component-based software development has emerged as a promising approach to developing complex systems, via an approach of composing smaller, independently developed components into larger assemblies. This approach offers means to increase software reuse, achieve higher flexibility and deliver shorter time-to-market by reusing existing component, such as off-the-shelf components. Component-based software development is realised in a number of different ways, e.g., through model-based development or service-oriented computing.

Safety critical systems, like many other domains, may benefit from the flexibility offered by component-based software development. However, to be applicable to safety critical systems, component-based development must directly support modelling and analysis of key non-functional concerns, such as availability, reliability, and the overall failure behaviour of the system, in order to deliver a

B. Buth, G. Rabe, T. Seyfarth (Eds.): SAFECOMP 2009, LNCS 5775, pp. 215–228, 2009.

system that is acceptably safe (e.g., to certifying authorities). The last concern is particularly challenging to deal with: a system can fail in many ways. It may be the case that a system failure arises due to failures of individual or a small number of components. Identifying the *source and likelihood* of system failure is of substantial importance to developers and safety engineers, so that they can be sure that they have appropriately mitigated risks. Specifically, it is important that safety engineers and system developers be able to understand the consequences of individual component failures.

1.1 Current Techniques for Failure Analysis

Several approaches to failure analysis, and understanding overall system failure behaviour, have been investigated, including research on software testing (e.g., [4,5,8,9,10,13,14]) and system engineering (e.g., [2,3,6,12,15]). Largely, this body of work provides evidence that understanding system failure behaviour is more difficult than understanding specified (acceptable) behaviour. Failure analysis techniques based on software testing (especially fault-based testing and mutation analysis) include Interface Propagation Analysis (IPA) [14] and the Propagation Analysis Environment (PROPANE) [5]. Both IPA and PROPANE studied propagation behaviour at the code level. There were also studies of propagation in terms of software architecture, e.g., [10]. Most of the research from the software testing perspective focused on the study of the propagation of data error, which occurs homogeneously — "for a given input, it appears that either all data state errors injected at a given location tend to propagate to the output, or else none of them do" [9]. In practice, the failure propagation behaviour of software components may become much more complex when considering failures caused by hardware components.

There are also approaches to analysis of failure propagation behaviour from the system engineering perspective. Perhaps the most well known approach is the classical safety engineering technique *Failure Modes and Effects Analysis* (FMEA) [1], which is a manual process for identifying the failure modes of a system starting from an analysis of component failures. Generally, the process of failure analysis consists of several activities: identifying failures of individual components, modelling the failure logic of the entire system, analysing a failure's effect on other components, and determining and engineering the mitigation of potential hazards.

In safety engineering, developers and engineers general model and analyse potential failure behaviour of a system as a whole. With the emergence of component-based development approaches, investigations began exploring component oriented safety analysis techniques, mainly focusing on creating *encapsulated* failure propagation models. These failure propagation models describe how failure modes of incoming messages (input failure) together with internal component faults (internal failure) propagate to failure modes of outgoing messages (output failure). Failure Propagation Transformation Notation (FPTN) [3] was the first approach to promote the use of failure propagation models. Other relevant techniques are Hierarchically Performed Hazard Origin and Propagation

Studies (HiP-HOPS) [12] and Component Fault Trees (CFT) [6]. For specific component-based specification languages, the later two techniques allow tool-supported and automated generation of a safety evaluation model. A limitation of these safety analysis techniques is their inability to handle cycles in the control- or data-flow architecture of the system; cycles, of course, appear in most realistic systems. Fault Propagation and Transformation Calculus (FPTC) [15] was one of the first approaches that could automatically carry out failure analysis on systems with cycles by using fixed-point analysis.

This paper focuses on failure propagation behaviours at the architecture level, in which the components may be hardware or software components. Based on our experience, we found that existing failure analysis techniques have a number of limitations, in particular:

- FMEA and FPTN generally provide manual or non-compositional analysis. Such analysis is expensive, especially in a typical component-based development process, because if changes are made to components, the failure analysis has to be carried out again, and previous analysis results will be invalidated.
- FPTC does not provide facilities for quantitative analysis, particularly in terms of determining the *probability* of specific failure behaviours. Such quantitative analysis can help to provide more fine-grained information to help identify and determine suitable (cost-effective) mitigation to potential hazards.

By providing an extension of FPTC technique, we are aiming to overcome the limitations we found in existing system engineering analysis techniques.

1.2 Contribution and Structure of the Paper

In this paper, we propose a safety analysis technique, *failure propagation and transformation analysis* (FPTA), which follows the direction of FPTC [15]. The FPTA method integrates an automated failure analysis algorithm presented in [15], and it also allows the application of model checking technique as provided by the PRISM[1] model checker [7]. The approach is therefore a probabilistic safety analysis technique for component-based system development.

The structure of the paper is as follows. We begin by presenting background material. We introduce the failure analysis technique in detail and outline its underlying theory, explaining how FPTC [15] is integrated with probabilistic model checking. Finally, we demonstrate the analysis method on an illustrative safety-related application.

2 Background

The theory and techniques of FPTC were initially introduced in [15], and the implementation of a supporting standards-compliant and open-source tool was presented in [11]. We will briefly describe the modelling and analysis techniques of FPTC in the following section.

[1] http://www.prismmodelchecker.org/

2.1 Failure Modelling

FPTC is based on FPTN [3], and is applied to a model of system architecture. In this approach the failure behaviours of *both* components and connectors are determined and modelled. FPTC takes the view that connectors between components are communication protocols, and because a communication protocol also has its own potential failure behaviour, the protocols in the model must be treated identically to the components of the system – i.e., their failures are also modelled.

Components and connectors can (in terms of failure) behave in only a few ways [15]. They can introduce new types of failures (e.g., because of an exception or crash), or may propagate input failures (e.g., data that is erroneous when it arrives at a component remains erroneous when it leaves the component), or transforms an input failure into a different kind of failure (e.g., data that arrives late may thereafter arrive early). Finally, a component may correct or mask input failures that it receives.

Failure responses of a component to its input can be expressed in a simple language based on patterns. For example, the following expressions denote examples of failure propagation and transformation behaviours for a trivial single-input single-output component: an *omission* fault at the input may propagate through the component, but a *late* fault is transformed to a *value* fault at the output.

$$\begin{array}{rcll} omission & \longrightarrow & omission & (failure\ propagation) \\ late & \longrightarrow & value & (failure\ transformation) \end{array}$$

A typical component will have its failure behaviour modelled by a number of clauses of this form, and the effect is its overall *FPTC* behaviour.

2.2 FPTC Analysis Technique

To represent the system as a whole, every element of the system architecture – both components and connectors – is assigned FPTC behaviour. Each model element that represents a relationship is annotated with sets of *tokens* (e.g., omission, late). The architecture as a whole is treated as a token-passing network, and from this the maximal token sets on all relationships in the model can be automatically calculated, giving us the overall failure behaviour of the system. This calculation resolves to determining a fix-point [15]. For details of the algorithm, see [11]; for an argument that the fix-point calculation must ultimately terminate, see [15]. Examples describing the use of FPTC in a number of domains, including for analysis control logic and FPGAs, appear in [11].

2.3 Analysis of FPTC

FPTC overcomes the problem of handling cyclic data- and control-flow structures in a system architecture by using fix-point calculations. As well, experience from a number of case studies suggests that it can be integrated into a system

design process, thus potentially reducing safety engineering overheads. FPTC nevertheless has some limitations which make it difficult to extend directly to richer forms of analysis, particularly probabilistic analysis. We summarise these limitations by example.

Example I: internal failures. Consider a simple system with a component (e.g., a hardware sensor) that may have an *internal* power failure. When a power failure occurs, there will be no output (i.e., an omission failure) from the component, no matter what its inputs are. This can be modelled implicitly in FPTC as shown in Equation 1[2].

$$input.* \longrightarrow output.omission \tag{1}$$

This does not explicitly model the fact that there has been an internal failure in the component. This is not a problem for standard FPTC, but if we desire to extend FPTC to probabilistic analysis, we encounter difficulties: suppose an internal omission failure occurs with probability 0.01. To model this, we need to distinguish the case where an internal failure arises (described in Equation 1) from the case where an omission failure is *propagated* by the component (i.e., the omission failure occurs elsewhere in the system). This requires the addition of a new FPTC equation.

$$\begin{aligned} input.omission &\longrightarrow output.omission \\ input.* &\longrightarrow output.omission \end{aligned} \tag{2}$$

The first line states that an omission fault on input leads to an omission fault on output. The second line indicates that any fault on input leads to an omission fault. But the first line is an instance of the second, and according to the definition of FPTC analysis in [15], is removed from calculations. But it is explicitly necessary in order to support probabilistic analysis, because we must be able to distinguish internal from external omission failure.

Unlike other techniques, such as HiP-HOPS and CFT, the FPTC technique targets software systems where component failures are only triggered by inputs. It is thus lacking in its support for modelling internal failure in the process of integrated software/hardware design and assessment. Overall, input and output failures are generally straightforward to identify, but a failure model given strictly in terms of inputs and outputs may be insufficient to adequately capture system failure behaviour, particularly when probabilities are involved.

Example II: non-determinism. Suppose we have a situation where a component may not receive the inputs it requires (i.e., an omission failure on input), and as a result, the component will, half of the time, generate no output, and the other half of the time will generate the wrong output (i.e., a value failure). In FPTC, this component can be partly modelled as in Equation 3.

$$\begin{aligned} input.omission &\longrightarrow output.omission \\ input.omission &\longrightarrow output.value \end{aligned} \tag{3}$$

[2] $*$ Indicates any input.

This FPTC model is not well-formed according to [15], because the algorithm assumes that failure behaviours on outputs are deterministic. Such behaviours cannot be automatically analysed with the existing algorithm, but being able to represent such behaviours is essential in order to support probabilistic analysis.

Summary. Overall, FPTC addresses one significant limitation of other safety analyses – handling cycles in architectures – but is still insufficient. The limitations we have identified are all related to how failures are modelled currently in FPTC; the coarse nature of failure modelling in FPTC (particularly, the inability to represent internal failures and non-deterministic failure behaviour on output) makes it difficult to extend to probabilistic analysis. In the next section, we propose an extension to FPTC models that supports probabilistic modelling, and that eliminates these problems.

3 Probabilistic Modelling Extensions to FPTC

In this section we present an extension to FPTC for supporting probabilistic modelling and that address the concerns presented above. We call this extension *failure propagation and transformation analysis* (FPTA).

3.1 Probability Property

The first limitation presented with FPTC was the inability to explicitly model internal failures in components (or connectors); this limitation is particularly important to resolve in order to describe the uncertainty of a component's transitive behaviours when considering the impact of internal failures. The overall effect of this limitation is that internal failures are masked.

To alleviate this limitation in FPTA, we extend the model of FPTC failure behaviour, by providing richer, more expressive means for modelling inputs and outputs, the *mode* that inputs and outputs are in, and the probability associated with each mode. We now explain this more precisely.

3.2 Transitive Behaviour Model

Components in FPTA (as in FPTC) are the principal processing objects of the executing system, and connectors are interaction or communication mechanisms for components. In most realistic system architectures, a component may have multiple input ports and output ports; a port is the point of interaction between component and connector. The values placed on an output port can be calculated via a function that takes all input values into account. This is called the *transition function* in FPTA. For example, a component with n input ports and m output ports will have m transition functions. An instance of a transition function can be written as:

$$\{input_1.fault, \ldots, input_n.fault\} \longrightarrow output_x.fault, probability \quad (4)$$

Mode is a term used in FPTA to describe the state of the contents of an input or output port. Since an output of a component may have many modes, the transition function of an output can have many instances of its possible modes. We use a tuple (*mode*, *probability*) to describe each mode of an in/output port. We call this tuple a *token*.

3.3 System Model

Connectors in an architectural model of a system are the links between components. In FPTA, we provide a different semantics to connectors than in FPTC: they are an abstraction that does not have any failure transformation behaviour. Specifically, they propagate whatever they receive from an input port to an output port. Thus, failure behaviour is modelled exclusively on components; this, as we will see, simplifies the probability calculations.

Modes are propagated by connectors at run-time one at a time. Mathematically, what is propagated by a connector is a set of all possible input or output modes. To model a connector, we define the contents propagated by a connector as a collection of tokens. Formally, this is $\{token_1,\ token_2, \ldots, token_n\}$ if there are n possible modes that can be propagated.

Based on this definition, the tokens of a connector should satisfy the following expression.

$$\sum_{i=1}^{n} token_i.probability = 1 \tag{5}$$

The system model is thereafter constructed by connecting the models of all its components in the same way as is done in traditional FPTC.

So far, we have explained how we model the system. Next, we will revisit the limitations of the FPTC modelling language.

3.4 Revisiting Limitations of FPTC

Given FPTA as presented in the previous subsections, we now show that it overcomes the limitations of FPTC discussed previously.

Consider the example presented in Section 2.3. Assume that a power failure occurs with probability 0.01. The failure behaviour of this component can now be expressed as:

$$\begin{array}{rcl}
input.omission & \longrightarrow & output.omission\ , 0.0001 \\
input.value & \longrightarrow & output.omission\ , 0.0001 \\
input.normal & \longrightarrow & output.omission\ , 0.0001 \\
input.omission & \longrightarrow & output.omission\ , 0.9999 \\
input.value & \longrightarrow & output.value\ , 0.9999 \\
input.normal & \longrightarrow & output.normal\ , 0.9999
\end{array} \tag{6}$$

The example in Section 2.3 described the case where a component may have different ways of reacting to the same failure on input. In particular, when there is an omission failure of input, the component will half of the time generate

no output (omission) and half of the time will generate the wrong value (value failure). This failure behaviour can be described as follows.

$$\begin{array}{rcl} input.omission & \longrightarrow & output.omission\ , 0.5 \\ input.omission & \longrightarrow & output.value\ , 0.5 \\ input.value & \longrightarrow & output.value\ , 1 \\ input.normal & \longrightarrow & output.normal\ , 1 \end{array} \tag{7}$$

Again, there is essential complexity that arises in modelling failure behaviour when probabilities are introduced.

3.5 Analysing the System

Once the system model is constructed by connecting models of components, we *"execute"* the model by using an algorithm similar to FPTC. Tokens in FPTA consist of two elements: a mode and its probability. The technique to deal with the computation of the modes is as same as the fix-point technique used in FPTC. The law of total probability is used to calculate the probability associated with each mode.

In FPTA, if there are n possible modes that can be transitioned to a particular failure of an output, the law of total probability says:

$$P(output.fault) = \sum_{i=1}^{n} P(output.fault|input.mode_i)P(input.mode_i) \tag{8}$$

In this formula, the probability values of input modes are calculated by previous computations, and the conditional probability $P(output.fault|input.mode_i)$ is modelled in the instance of a transition function in the model of the component. At the beginning of the *"execution"*, engineers provide an initial set of probability values for modes, and then the calculation is carried out automatically until the execution stops. Because the failure set is finite, the computation will be guaranteed to reach a fix-point.

We now provide several examples. Given a component, assume that there are two possible modes in which the component can fail, namely *value* (v) and *omission* (o). As well, there is a default non-failure mode (*normal* (n)). First, we consider the case where the component has one input port and one output port. Example transitive behaviours and their probabilities (identified by a domain expert) are listed in Table 1.

Table 1. The probability of possible transitions

Input Modes	**Output Modes**		
	normal (n)	value (v)	omission (o)
n	0.89	0.1	0.01
r	0	0.99	0.01
nr	0	0	1

According to the data given by Table 1, the transition model of the component is:

$$\begin{array}{rl} input.n & \longrightarrow output.n\ , 0.89 \\ input.n & \longrightarrow output.r\ , 0.1 \\ input.n & \longrightarrow output.nr\ , 0.01 \\ input.r & \longrightarrow output.r\ , 0.99 \\ input.r & \longrightarrow output.nr\ , 0.01 \\ input.nr & \longrightarrow output.nr\ , 1 \end{array} \tag{9}$$

We can easily determine the set of tokens for the input port on the component; this is $\{(n, 0.9),\ (v, 0.05),\ (o, 0.05)\}$. And from this, we can calculate the tokens, including the modes and their probability values, for the output port by computing the probability of every possible mode at the output port. For example,

$$\begin{aligned} P(output.n) &= P(input.n) \cdot P(output.n|input.n) \\ &\quad + P(input.v) \cdot P(output.n|input.v) \\ &\quad + P(input.o) \cdot P(output.n|input.o) \\ &= 0.9 \times 0.89 \\ &= 0.801 \end{aligned} \tag{10}$$

After carrying out a similar calculation for all other possible modes, the token set at the output port is:

$$\{(n, 0.801),\ (v, 0.1395),\ (o, 0.0595)\} \tag{11}$$

In a larger system, if this component is connected to another, we would take the token set, $\{(n, 0.801),\ (v, 0.1395),\ (o, 0.0595)\}$, and use them as input tokens of a component connected to this output port. This process would carry on until the calculation reaches a fix-point and stops.

3.6 Model Checking

Our component-oriented approach for analysing failure behaviour focuses on the transitions between modes of a component. Since we introduced a probability property into the failure model, we need a formal mechanism to verify the probabilistic model. Probabilistic model checking [7] is a suitable mechanism to use for verification in this situation. Probabilistic model checkers encode system models using Markov chains; in this sense, they encode the probability of making a transition between states instead of simply the existence of a transition. The probabilistic model checking process is an automatic procedure for establishing if a desired property holds in a probabilistic system model. We exploit probabilistic model checking – and, in particular, PRISM – to accomplish three purposes.

1. It can be used to formally verify the probabilistic model. Since the probability property was introduced to describe the failure behaviours of a component, the model checker can help to check criteria that the probability values must satisfy.

2. The output token set of a component can be calculated by the model checker during the analysis. When the model becomes complex (e.g., when a component has three or more input ports, or there are more than three possible modes at each port), the calculation of output token set will be very difficult without automated tool support.
3. There is also a desire that safety engineers and system developers can easily determine how critical a component is to the entire system. Ideally, it should be possible for the relationship between the failure behaviour of a component, and the entire system, can be visualised.

In order to use the PRISM probabilistic model checker, we have to precisely define the state of a component as a finite state model; note that FPTC abstracts away from internal state and represents failures as observable external behaviour. The state of a component can be formally defined by the modes of its input and output ports because they can be observed and measured directly.

Once the transitive behaviour model is expressed as a state machine, then it is very straightforward to express the model in the PRISM input language; space limitations prevent us from presenting this simple transformation. Properties can then be checked against the model by the PRISM model checker. Any counter-examples identified by the PRISM model checker can easily be mapped back to the state machine, and then manually reflected against the transitive behaviour model. Full automation of this process would be of benefit, and we are investigating the use of model transformation technology (and automated traceability management) to support this.

4 Example

We now illustrate the overall analysis process via a small, yet realistic example. The system for case study is a piece of a safety-critical control system. There are 6 components in the system —`U1` to `U6`: Component `U1` outputs the absolute value of the input; `U2` outputs the product of two inputs; `U3` is an amplifier (3 times); `U4` generates a constant value; `U5` gives the minimum value of two inputs; and `U6` outputs the division of two inputs. Figure 1 shows its architecture. We applied the probabilistic failure analysis technique to a logic unit used in control systems.

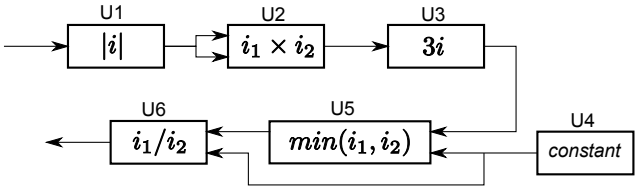

Fig. 1. Architecture of logic unit

The application is a software system (though it is normally deployed with supporting hardware, and hardware failures may lead to software failures, and vice versa). Using HAZOPs and guide-words, we identified two kinds of failure modes, *value* (v) and *omission* (o), and a default non-failure mode, *normal* (n).

Based on the knowledge of the transitive behaviours of components in the system, we modelled the components in the architecture one by one. For example, Equation 12 is the model of component `U5`.

$$\begin{array}{ll}
\{input.n,\ input.n\} & \longrightarrow output.n\ , 0.89 \\
\{input.n,\ input.n\} & \longrightarrow output.v\ , 0.1 \\
\{input.n,\ input.n\} & \longrightarrow output.o\ , 0.01 \\
\{input.n,\ input.v\} & \longrightarrow output.v\ , 0.99 \\
\{input.n,\ input.v\} & \longrightarrow output.o\ , 0.01 \\
\{input.n,\ input.o\} & \longrightarrow output.o\ , 1 \\
\{input.v,\ input.v\} & \longrightarrow output.v\ , 0.99 \\
\{input.v,\ input.v\} & \longrightarrow output.o\ , 0.01 \\
\{input.v,\ input.o\} & \longrightarrow output.o\ , 1 \\
\{input.o,\ input.o\} & \longrightarrow output.o\ , 1
\end{array} \tag{12}$$

Based on these models of the components and the architecture of the system, we carried out several experiments. The first experiment examines the probability of the component `U6` outputting *normal* values if the input of component `U1` is *normal*. In this case, the input token set of component `U1` is $\{(n, 1),\ (v, 0),\ (o, 0)\}$. Applying the probabilistic FPTA technique, the automatically calculated output token set of component `U6` is:

$$\{(n, 0.4423),\ (v, 0.4898),\ (o, 0.0679)\}$$

Similarly, we calculated the case where the input tokens are: $\{(n, 0),\ (v, 1),\ (o, 0)\}$ and $\{(n, 0),\ (v, 0),\ (o, 1)\}$. The calculated output tokens are

$$\{(n, 0),\ (v, 0.9321),\ (o, 0.0679)\}$$

and

$$\{(n, 0),\ (v, 0),\ (o, 1)\}$$

Once we have obtained the input and output token sets for the individual components, we can model the entire logic unit, consisting of components `U1` to `U6`, as follows:

$$\begin{array}{lll}
input.n & \longrightarrow & output.n\ , 0.4423 \\
input.n & \longrightarrow & output.v\ , 0.4898 \\
input.n & \longrightarrow & output.nr\ , 0.0679 \\
input.v & \longrightarrow & output.v\ , 0.9321 \\
input.v & \longrightarrow & output.o\ , 0.0679 \\
input.o & \longrightarrow & output.o\ , 1
\end{array} \tag{13}$$

The result of our first small example shows that the FPTA technique can be applied hierarchically, which allows the decomposition of the probabilistic evaluation based on the system architecture.

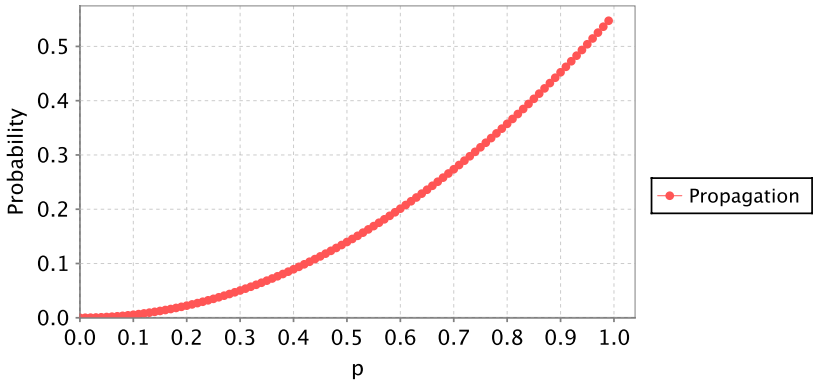

Fig. 2. Relationship of Component `U4` to Overall Unit

Our second example is a variant on the first, and explores how changing a component in an architecture can affect the overall system failure behaviour. In particular, we show how we can use FPTA to explore different choices in modelling. Suppose that we are unhappy with the performance of `U4` (a constant generator) in our example, and desire a design for `U4` that provides a better error rate. Through experiment, we want to understand the effect of introducing a better-performing component on the entire system.

In the experiment, we model component `U4` as follows:

$$\begin{array}{ll} input.n & \longrightarrow output.n\ , p \\ input.n & \longrightarrow output.v\ , 1-p \end{array} \tag{14}$$

In the model (Equation 14), we introduce a variable p which is the conditional probability that the component generates a *normal* output. We transform the failure model in PRISM and set up a series of experiments in which value p is increased from 0 to 1 with step-size of 0.01. Figure 2 shows the results of experiments, where the X-axis indicates the trend for p and the Y-axis is the probability that the entire system outputs *normal* data (i.e., the output of `U6` is *normal*), assuming that the input of the system (i.e., input of `U1`) is *normal*.

We can conclude from Figure 2 that there isn't a linear relationship between the non-failure rate of component `U4` and the non-failure rate of the entire system; the better the component `U4` (i.e., the smaller its error rate) is, the better the overall failure behaviour of the entire system.

In addition, suppose that we want the overall non-failure rate of the entire system to be not less than 0.5. From Figure 2 we observe that the non-failure rate of component `U4` should not be less than 0.95; in fact, the overall non-failure rate of the entire system is 0.504 if the non-failure rate of `U4` is 0.95 (i.e., the failure rate of `U4` is 0.05).

This example shows that FPTA can be applied to analyse the criticality of a component in the system, and to help to set up criteria for component selection; this is very important in a component-based development process.

5 Conclusions

We have presented a new technique for quantitative analysis of failure behaviour for systems, based on architectural models. The proposed technique enables the assessment of failure behaviour from the analysis of components of the system, and can assess the probability of system-level failures based on failures of components. The approach has been connected to a probabilistic model checker, which allows verification of the failure models, but also helps to calculate input and output token sets and helps in exploring the model. Importantly, the approach is compositional, and can be applied to individual components and collections.

Our transformation from the failure model to input used by the PRISM tool is currently carried out manually; this can potentially lead to errors in the PRISM input. In our experience, errors are often found by PRISM, but many of these could be avoided with an automated transformation from our failure models to PRISM. We are currently building a tool which is based on our previous work [11]. The idea is that once we model the failure behaviours of all components in the system architecture, we then transform the model to PRISM model using model transformation technology. We are also exploring using customised editors to visually represent feedback from PRISM on models of system architecture.

Acknowledgements

We thank Radu Calinescu (Oxford) for his help. This research was carried out as part of the Large-Scale Complex IT Systems (LSCITS) project, funded by the EPSRC through grant EP/F001096/1.

References

1. IEC 60812. Functional safety of electrical/electronical/programmable electronic safety/related systems, analysis techniques for system reliability - procedure for failure mode and effect analysis (FMEA). Technical report, International Electrotechnical Commission IEC (1991)
2. Fenelon, P., McDermid, J.A.: New directions in software safety: Causal modelling as an aid to integration. Technical report, High Integrity Systems Engineering Group, Dept of Computer Science, University of York (1992)
3. Fenelon, P., McDermid, J.A.: An integrated toolset for software safety analysis. The Journal of Systems and Software 21(3), 279–290 (1993)
4. Hiller, M., Jhumka, A., Suri, N.: An approach for analysing the propagation of data errors in software. In: Proceedings of 2001 International Conference on Dependable Systems and Networks DSN 2001, Göteborg, Sweden, July 2001, pp. 161–172. IEEE Computer Society, Los Alamitos (2001)

5. Hiller, M., Jhumka, A., Suri, N.: Propane: an environment for examining the propagation of errors in software. In: Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, pp. 81–85. ACM, New York (2002)
6. Kaiser, B., Liggesmeyer, P., Mäckel, O.: A new component concept for fault trees. In: Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software, SCS 2003 (2003)
7. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM: Probabilistic symbolic model checker. In: Field, T., Harrison, P.G., Bradley, J., Harder, U. (eds.) TOOLS 2002. LNCS, vol. 2324, pp. 200–204. Springer, Heidelberg (2002)
8. Li, B., Li, M., Ghose, S., Smidts, C.: Integrating software into PRA. In: Proceedings of 14th International Symposium on Software Reliability Engineering, ISSRE 2003, Denver, CO, USA, November 2003, pp. 457–467 (2003)
9. Michael, C.C., Jones, R.C.: On the uniformity of error propagation in software. In: Proceedings of 12th Annual Conference on Computer Assurance (COMPASS 1997), pp. 68–76 (1997)
10. Nassar, D.E.M., Abdelmoez, W., Shereshevsky, M., Ammar, H.H., Mili, A., Yu, B., Bogazzi, S.: Error propagation analysis of software architecture specifications. In: Proceedings of the International Conference on Computer and Communication Engineering, ICCCE 2006, Kuala Lumpur, Malaysia (May 2006)
11. Paige, R.F., Rose, L.M., Ge, X., Kolovos, D.S., Brooke, P.J.: Automated safety analysis for domain-specific languages. In: Proceedings of Workshop on Non-Functional System Properties in Domain Specific Modeling Languages, co-located with 11th International Conference of Model Driven Engineering Languages and Systems, MoDELS 2008. LNCS, vol. 5421, Springer, Heidelberg (2008)
12. Papadopoulos, Y., McDermid, J.A., Sasse, R., Heiner, G.: Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure. Reliability Engineering and System Safety 71, 229–247 (2001)
13. Voas, J.M.: Pie: A dynamic failure-based technique. IEEE Transaction of Software Engineering 18(8), 717–727 (1992)
14. Voas, J.M.: Error propagation analysis for COTS systems. IEEE Computing and Control Engineering Journal 8(6), 269–272 (1997)
15. Wallace, M.: Modular architectural representation and analysis of fault propagation and transformation. Electronic Notes in Theoretical Computer Science 141(3), 53–71 (2005)