Improving Requirements Specification for Communication Services with Formalised Use Case Models

Oksana Ryndina, Pieter Kritzinger
Data Network Architectures Group
Department of Computer Science, University of Cape Town
Rondebosch, 7701, South Africa
Email: {kryndina,psk}@cs.uct.ac.za
Telephone: +2721 650 3127, Fax: +2721 689 9465

***Abstract*— The challenging task of requirements specification for communication services has not been sufficiently addressed to date. The complexity of communication systems requires a formal approach to requirements capture and analysis, however at the same time the industry does not take well to convoluted formalisms. We suggest improving requirements specification by enhancing the approach that is most popular at the moment - use case modelling. We amend traditional use case models with a formal structure and semantics to make them suitable for automated verification. The enhanced use case modelling technique that we propose is called Susan (”S”ymbolic ”us”e case ”an”alysis) and it facilitates verification of use case models using symbolic model checking. We also developed a software tool called SusanX to construct, manipulate and analyse Susan models. The analysis feature of the tool is implemented using the NuSMV model checker. A number of generic properties for verification are built into SusanX, and the tool additionally allows the user to construct model-specific properties.**

I. Introduction

Software engineering of *communication services* has long been recognised as an especially challenging endeavor. This category of services includes telecommunication services, Internet services and hybrid services that span multiple network technologies. The difficulty in development arises from the complex nature of communication services, which includes characteristics such as concurrency, distribution and heterogeneity [1]. Ad hoc development of these systems is unacceptable and hence application of formal methods in the communications domain has been advocated by many [2][3]. However, while many formal techniques have been used for the behavioural design of communication services [4][5][6], specification of their requirements has not received much attention.

Our research focuses on enhancing *Requirements Specification (RS)* of complex systems, and communication services in particular. Literature suggests that convoluted formal methods do not establish well in the industry, while semi-formal techniques such as scenario-driven approaches are much more accepted by developers [7]. We attempt to bring together the strengths of both formal and semi-formal techniques by improving the RS approach that is most popular at the moment - *use case modelling* [8][9]. In our proposal, we amend traditional use case models with a formal structure and semantics to make them suitable for automated formal analysis. Formal analysis of use case models allows one to discover logical flaws and missing requirements early in the development cycle, and provides developers with much better insight into their models.

The enhanced use case modelling technique that we propose is called Susan (“S”ymbolic “us”e case “an”alysis) and it facilitates analysis of use case models using *symbolic model checking* [10]. We also developed a software tool called SusanX to construct, manipulate and analyse Susan models. To the best of our knowledge, our approach to improving use case modelling is unique.

The main objective of this paper is to introduce the Susan technique and demonstrate its advantages. The next section provides background on standard use case modelling. Section III explains Susan in some detail and introduces the SusanX tool. Section IV describes how we implemented formal model analysis with SusanX. In Section V we go through a simple example to demonstrate the proposed technique. The last section gives conclusions and suggestions for future work.

II. Use case modelling

The use case modelling approach was first presented by Jacobson [11], but now this technique is considered to be a part of the *Unified Modelling Language (UML)* [9]. Use case models specify functional requirements for a system in terms of scenarios of interaction between the system and its environment. The main elements of these models are *actors* and *use cases*. Actors are used to represent entities that interact with the system, while use cases define services that the system must provide. Diagrammatically, use cases are shown as bubbles, actors as stick figures and associations between the two are represented by connecting lines. An example of a use case diagram specifying some requirements for a corporate Voice over IP system is shown in Figure 1.

A use case can also be seen as a collection of scenarios of system use that have the same goal [9]. Hence, there are usually a number of different scenarios or flows through each use case. Use case diagrams are often supplemented by some

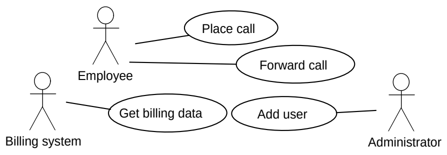


Fig. 1. Example of a use case diagram.

textual descriptions. For example, for each use case one can specify a priority, main flow, alternative flows, pre-conditions and post-conditions.

The main strengths of use case modelling are as follows.

(a) The approach is relatively simple and flexible.
(b) Use case models show *who* the stakeholders for the system are and *what* they require from the system, without showing *how* the system will be built.
(c) Stakeholders can understand use case models.
(d) Use case modelling is well-integrated into the Software Development Life Cycle (SDLC).

Despite these strengths of use case modelling, the approach suffers from several weaknesses that are listed below.

(a) Effective use case modelling is challenging.
(b) Textual use case descriptions often lack structure.
(c) Use case models and their supplementary descriptions are ambiguous.
(d) It is impossible to analyse use case models for correctness, completeness or consistency because they are not based on a formal syntax or semantics.

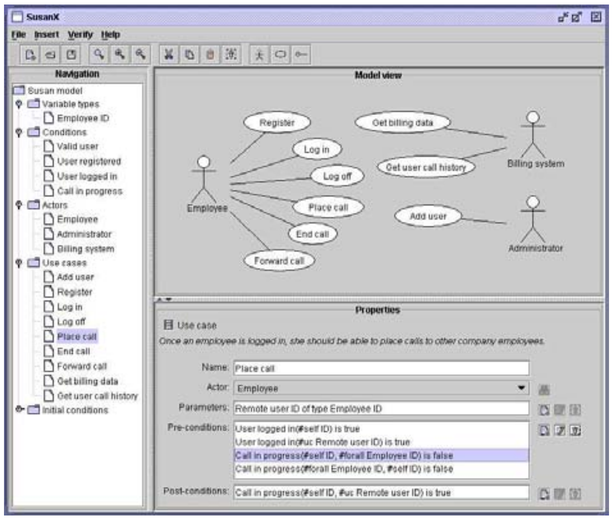
The weaknesses of use case modelling are especially serious in the context of communication services, where unambiguous specification of requirements is crucial and formal analysis can be very helpful. We propose the Susan technique to alleviate the above-mentioned drawbacks of use case modelling.

III. Susan modelling

The Susan technique comprises the following:

- **Susan metamodel** describes Susan modelling elements, their purpose, precise meaning and how they are related to each other.
- **Structural and semantic rules**
- **Verification support** is facilitated through a symbolic model checker called NuSMV [12]. A Susan model is translated to the NuSMV input language and then the NuSMV tool is used to perform verification.
- **SusanX** is a prototype software tool that we developed to allow one to construct, manipulate and verify Susan models. It interfaces with NuSMV to facilitate verification. Figure 2 shows the main interface of SusanX.
- **Guidelines** for constructing and analysing Susan models with SusanX are provided.

In Susan modelling, the system under consideration is treated as a "black box" and use cases are dealt with as autonomous and indivisible courses of action. In other words, we do not consider individual steps of use case flows. The diagram in Figure 3 illustrates the view on actor-system interaction taken by Susan, which is fundamental to the technique. The



Use case

Once an employee is logged in, s/he should be able to place calls to other company employees.

Name: Place call
Actor: Employee
Parameters: Remote user ID of type Employee ID
Pre-conditions: User logged in(#self ID) is true
User logged in(#uc Remote user ID) is true
Call in progress(#self ID, #forall Employee ID) is false
Call in progress(#forall Employee ID, #self ID) is false
Post-conditions: Call in progress(#self ID, #uc Remote user ID) is true

Fig. 2. The SusanX tool.

use case appears on the system boundary to show that it serves as a means of interaction between the actor and the system. The actor can call upon the system's services by *activating* use cases. The *global system state* is described by a set of *conditions* that change throughout model execution. Each use case is associated with a number of pre- and post-conditions. When a use case is activated, the state of the system is queried to determine whether the pre-conditions of the use case hold. If the pre-conditions are satisfied, the activation is *successful* and the post-conditions of that use case are used to alter the system state. During Susan model verification, all the possible interactions between the actors and the system are executed.

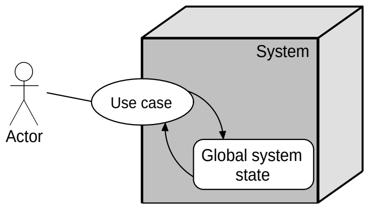


Fig. 3. Actor-system interaction in Susan.

The Susan metamodel and the structural and semantic rules for Susan are described next.

A. Susan metamodel

We took the fundamental building blocks of models from the standard use case approach and appended them with additional elements to facilitate construction of executable Susan models. The UML diagram in Figure 4 shows the Susan metamodel.

The aggregation relationships in Figure 4 show that a Susan model comprises four different types of elements: actors, use cases, conditions and variable types. For each modelling element the Susan metamodel prescribes a number of *properties*, which are similar to class attributes in the UML. The remaining element, variable, is auxiliary; it assists in defining properties for the main four elements.

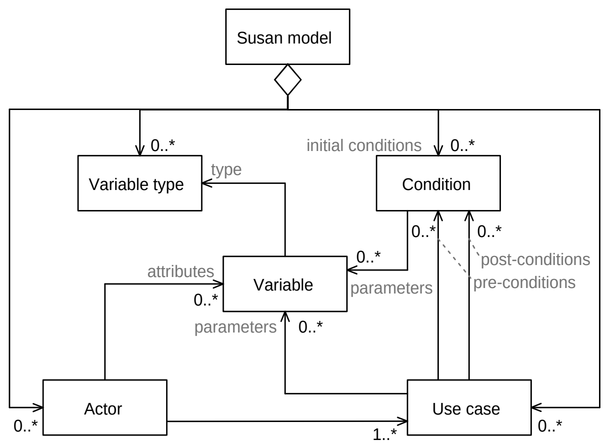


Fig. 4. Susan metamodel.

A Susan model consists of a use case diagram that shows actors, use cases and their associations. For each actor and use case in the diagram, textual properties are defined. Conditions and variable types do not have graphical representations; these elements are completely textual.

Actors: Susan defines two properties for an actor: a name and a list of *attributes*. Attributes describe an actor's particulars that the system needs to access in order to deliver services to that actor.

Use cases: In Susan, a use case has four properties: a name, a *parameter* list, pre-condition and post-conditions lists. Use case parameters describe information that is required by the system to provide the corresponding service. When a use case is activated, a value for each of its parameters needs to be passed to the system. A use case with values assigned to its parameters and the attributes of its associated actor is called a *use case instance*.

Pre-conditions indicate that certain things about the system state must hold in order for a use case activation to be successful. On the other hand, post-conditions describe how the system state changes after a successful activation of a use case.

Conditions: Conditions are used to describe the global state of the system and to declare use case pre- and post-conditions. Three properties are defined for a Susan condition: a name, a parameter list and a *truth-value*. A condition with values assigned to all its parameters is called a *condition instance*. A condition instance is either `true` or `false` at any given time during system execution; this is shown by its truth-value.

A number of *initial conditions* may be defined in a Susan model. These are condition instances that are `true` at the very beginning of system execution.

Variables and Variable types: Actor attributes, use case parameters and condition parameters are all variables. A variable in Susan has three properties: a name, a value and a *type*. Susan variables can only take on *symbolic values*, which are essentially string literals that can only be compared for equivalence. Two variables are equal if their values are set to identical string literals. Each variable is associated with a variable type, which is a finite set of symbolic values.

B. Susan structural and semantic rules

In addition to the metamodel a number of structural and semantic rules are necessary to completely explain how Susan models operate. The essentials of these rules are given below.

(a) **Completing a Susan model:** In a complete Susan model, properties of all the elements contained in the model are defined. The type property of all the actor attributes, condition parameters and use case parameters must be set to a variable type declared in the model. All the use case pre- and post-conditions must correspond to declared condition elements.

(b) **Defining pre- and post-condition properties:** In SusanX, when adding a pre- or post-condition to use case properties, the user must first make a selection from a list of existing conditions. Next, the user must match each of parameters for the chosen condition to one of the following: a parameter of that use case, an attribute of the actor associated with that use case or a symbolic value from the corresponding type. The user can also choose the *forall* option for such a parameter, in which case the pre- or post-condition must apply to all the values in the variable type for that parameter. Lastly, the user must specify the truth-value for the pre- or post-condition.
During system execution, actor attributes and use case parameters are assigned values non-deterministically. These values are then propagated to fill the pre- and post-condition parameters of the activated use case. Once the pre- and post-conditions have all their parameters assigned, pre-conditions can be queried against the current system state and post-conditions used to alter it.

(c) **Initial conditions:** Each initial condition must correspond to a declared condition element. All the parameters of initial conditions must be assigned.

(d) **Matching pre-conditions to post-conditions:** When a condition is used as a use case pre-condition, it must correspond to a post-condition for another use case or an initial condition.

IV. Verification of Susan models

Verification of Susan models is performed with the aid of the NuSMV tool, which is a symbolic model checker based on Binary Decision Diagrams (BDD). The NuSMV input language allows for description of finite state systems and specification of verification properties expressed in Computational Tree Logic (CTL) and Linear Temporal Logic (LTL). Susan defines all the verification properties in terms of CTL. An overview of the verification process is shown in Figure 5.

In order to support automated analysis, SusanX translates Susan models to NuSMV programs, passes them to the model checker that performs verification, and finally interprets the verification results for the user. A number of generic properties that can be used to verify any Susan model are built into SusanX. Additionally, SusanX allows the user to construct her own model-specific properties for verification using *property specification patterns* [13].

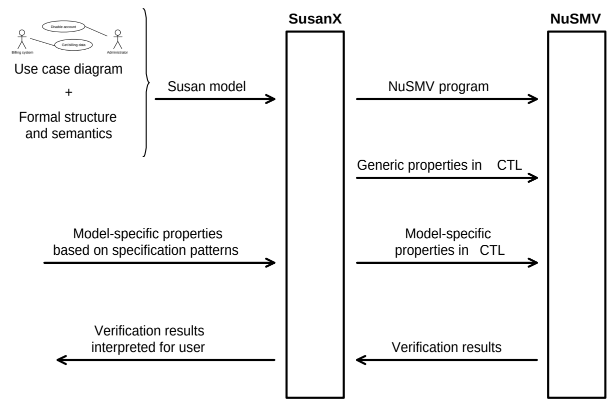

SusanX
NuSMV
Use case diagram
+
Formal structure and semantics
Susan model
NuSMV program
Generic properties in CTL
Model-specific properties based on specification patterns
Model-specific properties in CTL
Verification results interpreted for user
Verification results

Fig. 5. Verification of Susan models.

The details of our mapping of Susan models to NuSMV are outside of the scope of this paper. This section explains how verification for generic and model-specific properties is implemented in SusanX.

A. *Verification against generic properties*

SusanX provides generic verification that can be applied to any Susan model irrespective of the type of system being modelled. CTL specifications for the generic properties are built into the SusanX tool. These generic properties are used to analyse use cases for *liveness* and conditions for *reversability*.

Liveness of use cases: An informal definition of the liveness property is that "something good will always eventually happen" [14]. Susan defines three liveness categories for a use case: "Dead", "Transient" and "Live". SusanX analyses the model and places each use case instance into one of these categories.

(a) **Dead:** Successful activation of the use case instance is not possible. If all the instances of a use cases are "Dead", it is reported as a warning, because a use case that can never be successfully activated serves no purpose in the model.
(b) **Transient:** It is possible to successfully activate the use case instance a finite number of times. A typical example of this would be something that only happens once and is irreversible, for example "Dispose of call log data" can only be done once unless the log data is recoverable.
(c) **Live:** It is possible to activate the use case instance an infinite number of times.

Reversibility of conditions: SusanX analyses how condition instances change their truth-values throughout system execution. Each condition instance is placed into one of the following reversibility categories.

(a) **Constant:** The truth-value of the condition instance never changes, it remains the same as assigned initially.
(b) **Irreversible:** In this case the truth-value of the condition instance is changed once and then remains constant.
(c) **Finitely-reversible:** The condition instance changes its truth-value more than once, but still a finite number of times.
(d) **Reversible:** The condition changes its truth-value an infinite number of times.

Verification for liveness of use cases and reversibility of conditions generates a report that classifies each use case instance and condition instance according to the above-described categories. This report provides the user with insight into the behaviour of the system described by the model, as well as warns her of potential errors in the model.

B. *Verification against model-specific properties*

Verification against generic properties yields useful results, but because the generic properties are not model-specific this type of verification is limited. SusanX allows the user to define her own properties using property specification patterns. These patterns let one express simple properties for behavioural analysis without knowing the details concerning the underlying formalism, which is CTL in our case. We slightly tailored the specification pattern hierarchy developed by Dwyer *et al* to suit our specific needs for Susan model verification.

Each specification pattern contains one or more *pattern variables* that the user must substitute with valid values from the model being verified. A pattern variable is parameterised and may be `true` for some arguments and `false` for others. In SusanX, pattern variables can be constructed from: condition instances, use case instances and the logical operators `NOT` (`!`), `AND` (&), `OR` (`|`) and implication ($\rightarrow$). Once the user selects a pattern and fills in the pattern variables, SusanX generates the corresponding CTL specification property.

There are two main categories of specification patterns: *occurrence* and *order*. Our amended pattern hierarchy is shown in the following diagram.

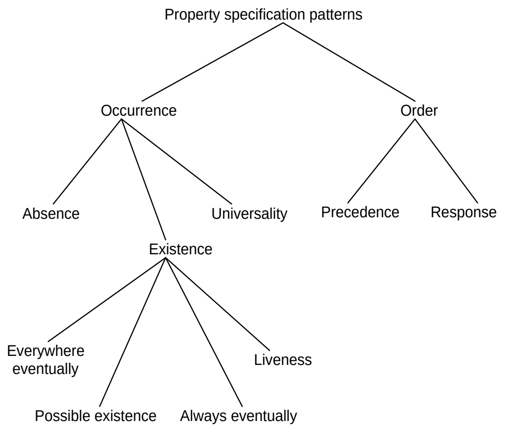


Fig. 6. Specification pattern hierarchy.

Occurrence: Occurrence patterns can be used to verify existence or absence of system states where a property holds.

(a) **Absence (Never):** *Safety properties* can be constructed using this pattern. An informal definition of a safety property is that "something bad will never happen" [14].
(b) **Universality (Globally):** This pattern can be used to express *invariants* for a model. An invariant is a property that must hold throughout the execution of the system.

(c) **Existence (Eventually):** If we are interested in reachability of certain system states, then this pattern can be used to construct properties for model verification. We extended the "Existence" pattern proposed by Dwyer *et al* and created four sub-categories of this pattern.
   - **Everywhere eventually:** Something will always eventually happen, no matter what execution path is taken.
   - **Possible existence:** It is possible for something to happen. In other words, the property may hold on some paths but not all the paths of execution.
   - **Always eventually:** No matter where in the system execution we are, something will always eventually happen. This pattern is a stronger variation of the "Everywhere eventually" pattern.
   - **Liveness:** Sometimes we want to ensure that at any time during the execution of the system, something will eventually become possible. This pattern is a stronger variation of the "Possible existence" pattern.

Order: Order patterns can be used to construct properties that verify a certain ordering of system states or events.

(a) **Precedence:** This pattern describes a dependency between two system states or events. It can be used to verify that one state or event always occurs before the other one.

(b) **Response:** This pattern is similar to the "Precedence" pattern but is used to verify that every cause must be followed by an effect rather than for every effect there must be a cause.

If verification for model-specific properties determines that a certain property is `false` then a counter-example trace of system execution is shown to the user. Such a trace consists of use case activations with the chosen values for each use case parameter and actor attribute. A trace may be finite or infinite. All infinite traces have a "loop", which is shown in the counter-example.

V. A SIMPLE EXAMPLE

In this section, we use a simple communication system example to illustrate the most important elements of Susan modelling and verification. We look at modelling functional requirements for a rudimentary corporate Voice over IP system. The use case diagram in Figure 7 shows the actors and use cases defined for the system.

The use of the Voice over IP system must be restricted to company employees only. The system administrator is responsible for maintaining a record of all valid users within the system. An employee who wishes to use the services of the system must first go through a registration process, during which a new account is created for her. A registered employee can log in to make calls and log off the system when finished. The company's billing system must interface with the Voice over IP system to get billing data and user call history.

We use the use case diagram from Figure 7 to construct a Susan model. We declare one variable type "Employee ID" and assign a finite set of test values to it. Next we declare conditions for the model: "Valid user", "User registered", "User logged in" and "Call in progress". For each of the actors

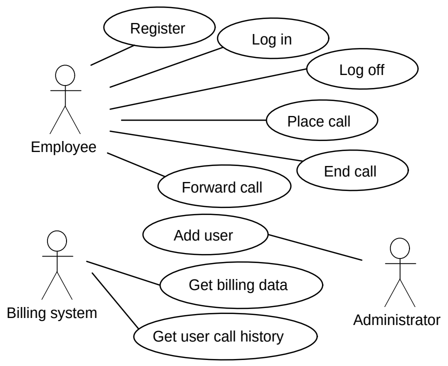


Fig. 7. Voice over IP system.

and use cases in the model, we add property definitions. Due to space limitations, we cannot include the complete description of the Susan model here.

For this system model, there are no initial conditions and hence we can begin verification. We first use the generic properties option for SusanX analysis and obtain results summarised in Figure 8.

Use case name	Liveness category
Add user	Transient
Register	Transient
Log in	Live
Log off	Live
Place call	Live
Forward call	Live
Get billing data	Live
Get user call history	Live

Condition name	Reversibility category
Valid user	Irreversible
User registered	Irreversible
User logged in	Reversible
Call in progress	Reversible

Fig. 8. Generic verification results.

The verification results show us that most of use cases fall into the "Live" category. "Add user" is one of the two use cases that are categorised differently, it is "Transient". We also observe that the "Valid user" condition is "Irreversible". Together, these two results tell us that once the administrator adds a user, that user will remain valid forever or rather until the end of system execution. What about employees who leave the company? These must not have access to the system's services, hence the system must provide a means of removing valid users. We correct this incompleteness in the model by adding a "Remove user" use case to the "Administrator" actor.

Note that we have an identical situation as above with the "Register" use case and the "User registered" condition. However, in this case if an employee decides to stop using the system then she can simply stop logging in, thus a deregistration service is not necessary.

The remaining results seem plausible - users can log in and off the system, calls get established and ended as required. We now use SusanX to formulate some model-specific properties that the model must satisfy. Below we show how these properties are constructed using specification patterns, and provide the corresponding verification results.

(a) **Only registered users should be allowed to participate in calls.** We use the “Universality” pattern to express this property:

Globally (Call in progress (a, b) → (User registered (a) & User registered (b)))

Verification shows that this property is `true`.

(b) **An established call will always be ended.** We use the “Response” pattern:

! Call in progress (a, b) *responds to* Call in progress (a, b)

Verification shows that this property is `true`.

(c) **A user cannot participate in more than one call at a time.** We use the “Absence” pattern to construct a set of properties that must all hold:

Never (Call in progress (a, b) & Call in progress (a, c))
Never (Call in progress (a, b) & Call in progress (c, b))
Never (Call in progress (a, b) & Call in progress (c, a))
Never (Call in progress (a, b) & Call in progress (b,c))

SusanX reports that this property does not hold, and produces a counter-example shown in Figure 9.

Step	Actor	Use case
1	Administrator()	Add user(a)
2	Employee(a)	Register()
3	Employee(a)	Log in()
4	Administrator()	Add user(b)
5	Employee(b)	Register()
6	Employee(b)	Log in()
7	Employee(a)	Place call(b)
8	Administrator()	Add user(c)
9	Employee(c)	Register()
10	Employee(c)	Log in()
11	Employee(c)	Place call(b)

Fig. 9. Counter-example trace.

In the last step of the counter-example trace, the call should not be established between “c” and “b”, since “b” is already on a call with “a”. More pre-conditions need to be defined on the “Place call” use case to check that the remote party is not engaged in a call with anybody else.

Once we corrected the discovered errors in the model, we ran verification against all properties once again. A number of such iterations were required to get the model to the desired state.

This simple example illustrates how to construct model-specific properties with specification patterns, and to interpret verification results for generic and model-specific properties.

VI. Conclusions and future work

The main objective of the work presented in this paper was to improve RS for complex systems such as communication services. We did this by developing the Susan technique based on use case modelling, and the supporting SusanX tool. Susan allows for creation of requirements models that are more complete, consistent and correct. SusanX provides the advantage of model verification without the user having to know the details of the underlying formalisms.

At this stage, Susan has not been applied to any large-scale systems and SusanX still needs to be extensively tested for usability and performance. Consequently, a broad case study and rigorous testing are our priorities for the near future. However, we believe that our project as it stands can already serve as valuable groundwork for further research in this area.

References

[1] X. Logean, F. Dietrich, and J.-P. Hubaux, “On Applying Formal Techniques to the Development of Hybrid Services: Challenges and Directions,” *IEEE Communications Magazine*, vol. 37, no. 7, pp. 132–138, July 1999.

[2] M.-P. Gervais and N. Ruffel, “Design of Telecommunication Service Based on Software Agent Technology and Formal Methods,” in *Proceedings of the IEEE Globecom’97*, 1997, pp. 1724–1728.

[3] J. P. Bowen and V. Stavridou, “The Industrial Take-up of Formal Methods in Safety-Critical and Other Areas: A Perspective,” in *FME’93: Industrial-Strength Formal Methods*, ser. Lecture Notes in Computer Science, J. C. P. Woodcock and P. G. Larsen, Eds., vol. 670. Springer-Verlag, 1993, pp. 183–195. [Online]. Available: citeseer.ist.psu.edu/bowen93industrial.html

[4] F. Belina and D. Hogrefe, “The CCITT-Specification and Description Language SDL,” *Computer Networks and ISDN Systems*, vol. 16, pp. 311–341, 1989.

[5] P. Zave, “Formal Description of Telecommunication Services in Promela and Z,” in *Proceedings of the Nineteenth International NATO Summer School*, 1999.

[6] L. J. Jagadeesan, C. Puchol, and J. E. V. Olnhausen, “A Formal Approach to Reactive Systems Software: A Telecommunications Application in ESTEREL,” *Formal Methods in System Design*, vol. 8, no. 2, pp. 123–151, March 1996.

[7] D. Amyot and A. Eberlein, “An Evaluation of Scenario Notations and Construction Approaches for Telecommunication Systems Development,” *International Telecommunications Systems Journal*, vol. 1, no. 24, pp. 61–94, September 2003.

[8] K. Bittner and I. Spence, *Use Case Modeling*. Addison-Wesley Publishers Ltd., June 2003.

[9] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language*. Addison-Wesley Publishers Ltd., 1999.

[10] K. McMillan, *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[11] I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*, 1st ed. Addison-Wesley Publishers Ltd., June 1992.

[12] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, “NuSMV: a new Symbolic Model Verifier,” in *Proceedings Eleventh Conference on Computer-Aided Verification (CAV’99)*, ser. Lecture Notes in Computer Science, N. Halbwachs and D. Peled, Eds., no. 1633. Trento, Italy: Springer, July 1999, pp. 495–499.

[13] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, “Property Specification Patterns for Finite-State Verification,” in *Proc. 2nd Workshop on Formal Methods in Software Practice (FMSP-98)*, M. Ardis, Ed. New York: ACM Press, 1998, pp. 7–15. [Online]. Available: citeseer.nj.nec.com/dwyer98property.html

[14] E. Kindler, “Safety and Liveness Properties: A Survey,” *Bulletin of the European Association for Theoretical Computer Science*, vol. 53, pp. 268–272, 1994. [Online]. Available: citeseer.nj.nec.com/59894.html

Oksana Ryndina obtained her Bachelor of Business Science degree with Honours in Computer Science from the University of Cape Town (UCT). She is currently pursuing an MSc in Computer Science with the Data Network Architectures Group at UCT. Her MSc research focuses on Requirements Specification and Formal Methods.