Improving Requirements Specification: Verification of Use Case Models with Susan

Technical Report CS04-06-00
Oksana Ryndina, Pieter Kritzinger
Data Network Architectures Group
Department of Computer Science, University of Cape Town
Rondebosch, 7701, South Africa

Abstract

Inadequate requirements specification is one of the main causes of software development project failure today. A major problem is the lack of processes, techniques and automated tool support available for specifying system requirements. We suggest a way to improve requirements specification methodology by enhancing the approach that is most popular at the moment - use case modelling. Despite their popularity, use case models are not formal enough for automated analysis. 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.

1. Introduction

It is fairly common knowledge that today only one out of every three software development projects is completed successfully. The latest CHAOS Surveys by the Standish Group [1] report that 15% of projects fail outright, and 51% are late, run over budget or provide reduced functionality. On average only 54% of the initial project requirements are delivered to the client. Inadequate specification of system requirements is considered to be one of the main causes for project failure.

What is it about requirements specification that developers find so challenging? One of the major issues is the lack of adequate processes, techniques and automated tool support available for specification of requirements. We thus set out in our research to enhance *Requirements Specification (RS)* methodology by improving the approach that is most popular at the moment - *use case modelling* [2][3]. The use case approach is well-suited for specifying functional requirements for software systems. Despite their popularity, use case models lack structure and exact semantics, which makes formal analysis of such models impossible. 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, presenting its strengths and weaknesses. Section 3 explains Susan in detail and introduces the SusanX tool. Section 4 describes how we implemented formal model analysis with SusanX. In Section 5 we go through a simple example to demonstrate the proposed technique. The last section gives conclusions and suggestions for future work.

2. Use case modelling

The use case modelling approach was first presented by Ivar Jacobson [8], but now this technique is considered to be a part of the *Unified Modelling Language (UML)* [3]. With use case models, one can 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. Sometimes use case bubbles are drawn inside a rectangle that symbolises the system boundary. An example of a use case diagram specifying some requirements for a corporate Voice over IP system is shown in Figure 1.

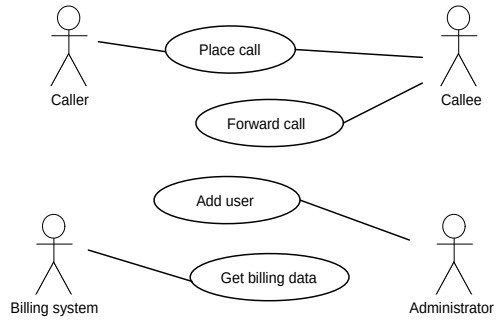

Figure 1. Example of a use case diagram.

In addition to defining a use case as a service required from the system, it can also be seen as a collection of scenarios of system use that have the same goal [3]. Hence, there are usually a number of different scenarios or flows through each use case - the main flow and several alternative flows.

Use case representation in diagrams is often supplemented by some textual descriptions. For example, for each use case one can specify a priority, main flow, alternative flows, trigger event, pre-conditions and post-conditions.

Three relationships can be used to connect use cases: *generalisation*, *include* and *extend*. With these relationships, one can show how a number of use cases share common behaviour. Actors can also be organised into generalisation hierarchies. If two actors are related by a generalisation relationship, the child actor inherits all the use case associations of its parent and "specialises" its parent by additionally having its own use case associations.

The main strengths of use case modelling are as follows.

(a) The approach is relatively simple and flexible.

(b) Use case models show the *what* and the *who* without the *how*. The goal of RS is to identify *who* the stakeholders for the system are and *what* they require from the system. Use case models are well-suited to capture this type of information without showing *how* the system needs to be built, which is a design concern.

(c) Stakeholders can understand use case models.

(d) Use case modelling is well-integrated into the Software Development Life Cycle (SDLC). Use case modelling naturally integrates into the software engineering process if the UML is used during the other development phases.

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

(a) Effective use case modelling is challenging. Although it is easy to learn the basics of use case modelling, effective use of this approach is not a simple task.

(b) Textual use case descriptions lack structure. There are no prescribed textual attributes that must be specified for a use case, neither are there formats set down for commonly used use case descriptions. Consequently, one can never be assured of the level of detail, type of content or presentation of use case descriptions.

(c) Use case models are ambiguous. Supplementary use case descriptions are usually given in natural language, which is inherently imprecise, making use case models ambiguous. Additionally, certain graphical elements of use case models are poorly defined [11].

(d) It is impossible to analyse use case models for correctness, completeness or consistency. Use case models are not based on a formal syntax or semantics, and as a result cannot be analysed in any formal way. This means that a use case model can only be analysed by hand for qualities such as correctness, consistency and completeness. Naturally, this becomes more difficult and unreliable as the amount of the information in the model increases.

We propose the Susan modelling technique to alleviate the weaknesses of use case modelling described above. Susan is described in detail in the next section.

3. Susan modelling

The Susan technique comprises the following:

- **Susan metamodel:** The metamodel describes Susan modelling elements, their purpose, precise meaning and how they are related to each other.
- **Structural and semantic rules:** These rules formally define the structure and semantics of Susan models.
- **Verification support:** Susan enables formal analysis of the constructed models by means of automated verification. A symbolic model checker called NuSMV [4]

is used to implement the verification. A Susan model is translated to the NuSMV input language and then the NuSMV tool is used to perform verification.

- **SusanX:** In order to test the feasibility of Susan and demonstrate the technique, we created a prototype software tool called SusanX. SusanX allows one to construct, manipulate and verify Susan models. It was implemented in Java and interfaces with the NuSMV model checker to facilitate verification. Figure 2 shows the interface of SusanX.

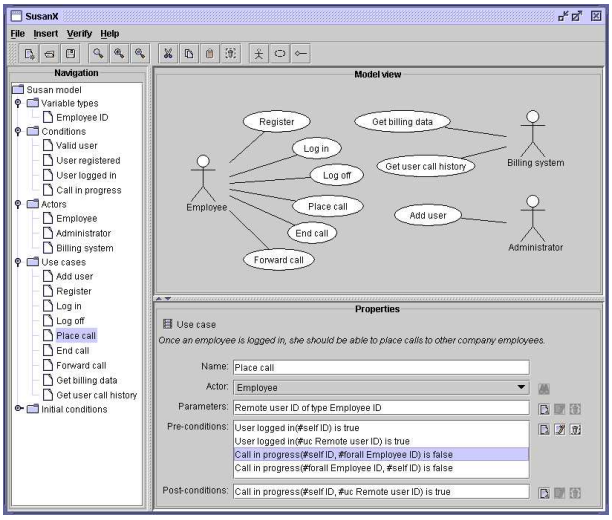

Figure 2. The SusanX tool.

- **Guidelines:** We provide a set of guidelines for constructing and analysing Susan models with SusanX. These guidelines also explain how to integrate Susan modelling into the rest of RS and SDLC.

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 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.

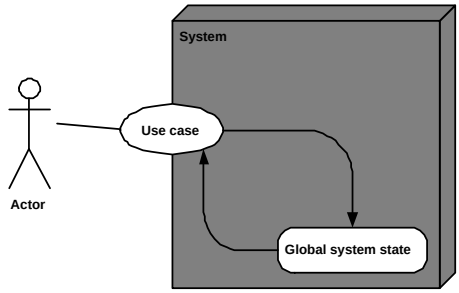

Figure 3. Actor-system interaction in Susan.

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

3.1. 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 following UML diagram shows the Susan metamodel. Classes are used to represent the modelling elements and associations denote relationships between the elements. The gray labels identify class roles, these show how one element can play different roles in different relationships. For example, the association between "Actor" and "Variable" should be interpreted as follows: "an actor has attributes that are variables".

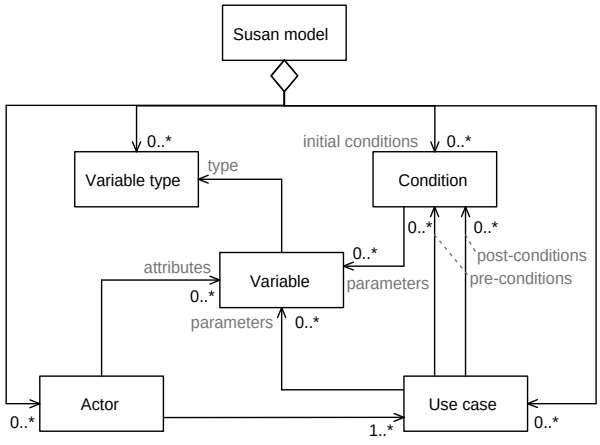

Figure 4. Susan metamodel.

The aggregation relationships in Figure 4 show that a Susan model comprises four different types of elements: ac-

tors, 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.

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. Each part of the Susan metamodel is described in detail next.

Actors: The Susan actor element is based on actors in the standard use case modelling. However, the actor-use case association is slightly more restrictive in Susan. The multiplicity for this association is "one to many" (1 to 1..*), which means that a use case can only be associated with one actor.

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. For example, a user of an online student services system may have one attribute - a student number.

Use cases: As in the standard approach, use cases represent functional system requirements. 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*.

The concept of pre- and post-conditions is not new in use case modelling, however it is not clearly defined in the standard approach. 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. If pre-conditions of a use case hold then once that use case is executed, its post-conditions will take effect. Susan treats conditions as modelling elements in their own right.

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 hold or 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.

From the diagram in Figure 4 it can be seen that Susan does not support relationships among use cases or actor generalisation relationships. We intend to integrate these additional features of use case models into the Susan technique in the near future.

3.2. Susan structural and semantic rules

The metamodel describes Susan modelling elements, their properties and how they are used in system models. However, 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) Adding elements to 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.

For SusanX users, we suggest first creating the main use case diagram. This defines actors and use cases for the model. Next, the user should add variable type definitions, followed by condition declaration. Once this is done, the user should proceed to defining properties for the actors and use cases that are already in the model.

(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 (prefixed by `#uc`), an attribute of the actor associated with that use case (prefixed by `#self`) or a symbolic value from the corresponding type. The user can also choose the `#foralll` 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. Figure 5 shows how the user defines pre- or post-condition properties in SusanX.

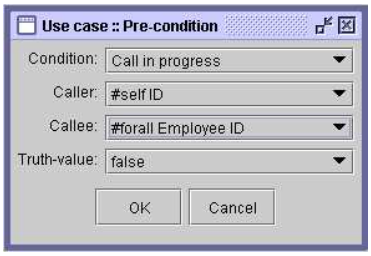

Figure 5. Adding a pre-condition in SusanX.

The concept of condition parameters in Susan is comparable to *formal* and *actual* parameters in programming languages. For example, in Java a method definition contains a formal parameter list, where the type of each parameter is specified. A method call supplies actual parameters to the method. Eventually, at runtime all the parameters are bound to actual values. In Susan, a condition declaration defines formal parameters for that condition and their variable types. When that condition is used as a pre- or post-condition for a use case, the user assigns each of the formal parameters to an actual parameter as described above. During the execution of the system model, all the possible use case activations are simulated. When a use case activation is simulated, the attributes of the associated actor and use case parameters are assigned actual values. These values are then propagated to fill the pre- and post-condition parameters of the 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: Initial conditions also form a part of the system model description in Susan. Each initial condition must correspond to a declared condition element. All the parameters of initial conditions must be assigned. We suggest that initial conditions are added to the model last, as the final step in preparing the model for verification.

(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. If a use case has an *unmatched* pre-condition, the possibility of that use case being successfully activated is eroded. SusanX performs a check for unmatched pre-conditions as part of ensuring that the model is ready for verification.

4. 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 the following diagram.

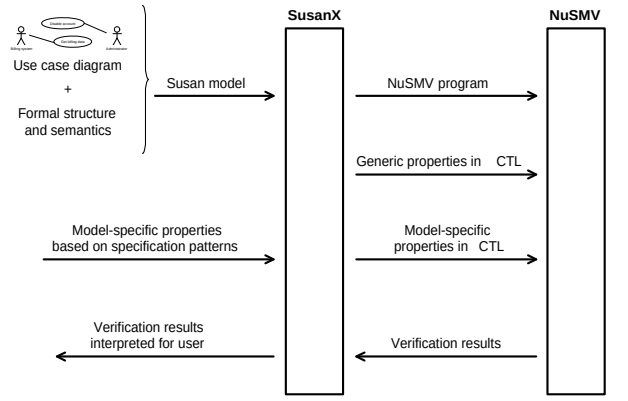

Figure 6. Verification of Susan models.

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*.

This section explains how Susan is mapped to the NuSMV language, as well as how verification for generic and model-specific properties is implemented in SusanX.

4.1. Translating Susan models to NuSMV

The NuSMV input language is designed to describe transition relations of finite state machines. During system execution, these transitions are used to determine valid evolution of the system state. The system state is represented by *state variables* in NuSMV. Only finite data types such as booleans, scalars and fixed arrays can be used for state variables. The basic structure of a NuSMV program is shown in Figure 7.

The program consists of one or more modules. The *main* module defines the entry point for system execution. All

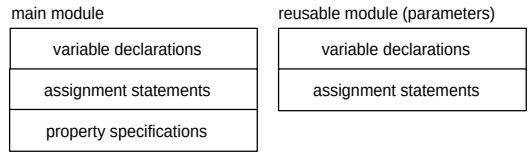

Figure 7. A NuSMV program.

modules except the main modules can be instantiated, hence they are called "reusable". Data is passed to modules by means of parameters. Parallel execution of modules can be achieved if they are instantiated as *processes*. Module instantiations are considered to be part of variable declarations in NuSMV.

Two main types of assignment statements are allowed in NuSMV: `init` and `next`. By assigning an initial value to a variable with `init` and then describing how this value changes in the next state with a `next`, one defines state transitions in a NuSMV program. NuSMV also allows non-deterministic assignments, which are useful in describing abstract behavioural models and representing uncertainties in a model.

When CTL model checking is used, property specifications consist of logical expressions constructed from the program state variables, CTL operators and quantifiers.

We map a Susan model to NuSMV in such a way that when the generated NuSMV program is executed, use cases are chosen randomly for activation and all variables within the model are assigned values non-deterministically. In this way, all the possible behaviours of the system are checked against the verification properties.

The structure outline of a NuSMV program generated from a Susan model is shown below.

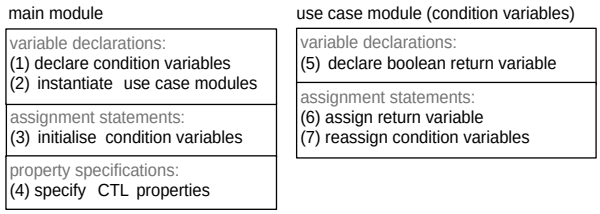

Figure 8. A Susan model in NuSMV.

Each condition instance in a Susan model is represented by a boolean variable in the NuSMV program (1). These condition variables are initialised in accordance with the initial conditions defined in the Susan model (3) and are used to represent the state of the system during execution.

For each use case instance, a module is declared within the NuSMV program. Inside a use case module, a boolean variable called `return` is declared (5) and assigned to `true` if the pre-conditions of the use case instance hold, and to `false` otherwise (6). Hence, the `return` variable can always be used to check whether the last use case instance activation was successful or not. If the activation of a use case instance is successful, then post-conditions are used to reassign condition variables moving the system into a new state (7). A use case module needs to access condition variables corresponding to the pre- and post-conditions of that use case instance. For this reason, the appropriate condition variables are passed to use case modules as parameters. Use case instance modules are initialised as processes (2) and during system verification, they are non-deterministically chosen for execution.

SusanX initially generates a ".smv" file without any CTL property specifications in the program. As verification is performed for various properties, the corresponding CTL specifications are inserted in the generated file (5).

4.2. 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" [9]. 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. "Transient" use cases can place a limitation on the system functionality and one should be sure that the use case irreversibility is actually intended.

(c) Live: It is possible to activate the use case instance an infinite number of times. It is expected that most use case instances would fall into this category.

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. It is expected that most conditions would fall into this category.

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.

4.3. 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.

Property specification patterns were first proposed by Dwyer *et al* in [5] and further supported by empirical studies [6]. The SAnToS Laboratory maintain an ongoing project for evolving these patterns, which is documented online [7]. Dwyer *et al* developed a system of specification patterns, which comprises a set of property specification patterns that are organised into a hierarchy showing relationships between different patterns. We tailored this system slightly 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. Pattern variables are *predicates* or in other words functions that yield a boolean value. 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 Figure 9.

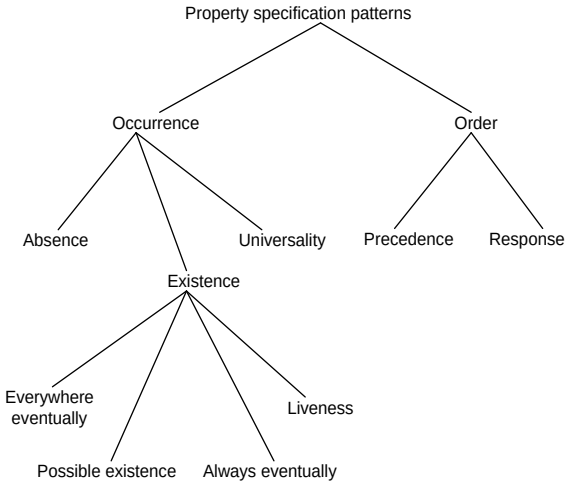

Figure 9. Specification pattern hierarchy.

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" [9].

(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. This pattern is closely related to the "Absence" pattern but while the "Absence" pattern is applied to negative properties, the "Universality" pattern applies to positive ones.

(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: Cause-effect relationships between system states or events can be expressed using this pattern. It 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. In the "Precedence" pattern causes may occur without subsequent effects, while in the "Response" pattern effects may occur without causes.

Figure 10 shows how model-specific properties are constructed in SusanX.

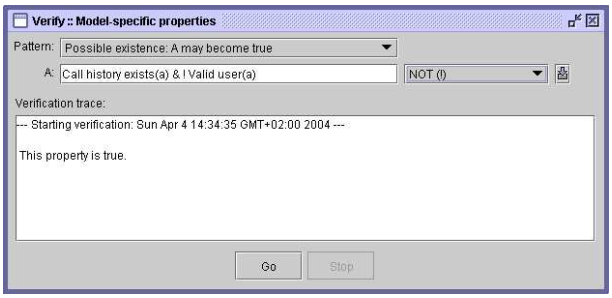

Figure 10. Constructing model-specific properties in SusanX.

In the above example, we use the "Possible existence" pattern to construct a property to check that call history may exist for users who are not currently valid. If "Call history exists" and "Valid user" conditions have parameters of "Employee ID" type, then during verification "a" will be replaced by values from that type. Note that both instances of "a" will be replaced by the same value.

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.

5. A simple example

In this section, we use a simple example to illustrate the most important elements of Susan modelling and verification. We look at modelling functional requirements for a simple corporate Voice over IP system. The main purpose of the system is to allow company's employees to make voice calls over the existing computer network. User authentication and call logging also need to be supported. The use case diagram in Figure 11 shows the actors and use cases defined for the system.

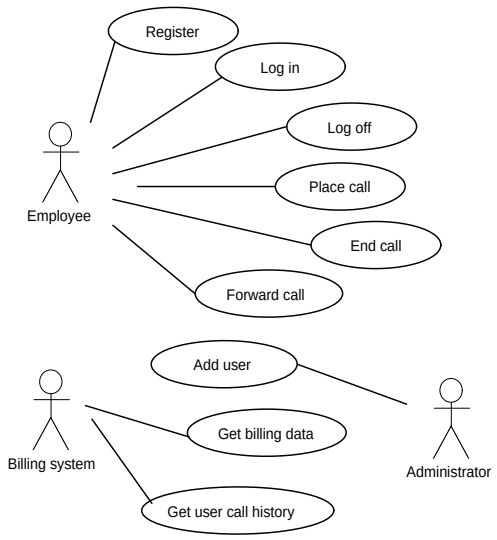

Figure 11. Voice over IP system.

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 11 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 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. Figure 12 shows the definition of the "Place call" use case.

For this system model, there are no initial conditions and

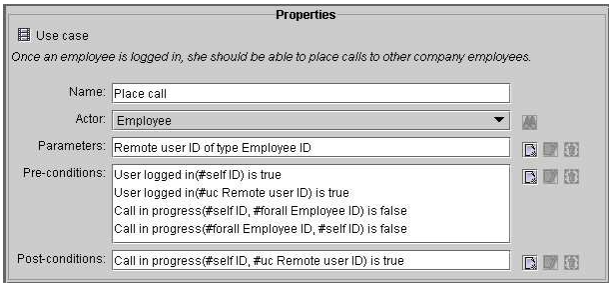

Figure 12. Place call use case.

hence we can begin verification. We first use the generic properties option for SusanX analysis and obtain results summarised in Table 1.

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

Table 1. Verifying generic properties

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) A user should not be able to establish a call with herself. Using the "Absence" pattern, we construct this property:

Never (Call in progress (a, a))

SusanX reports that this property is `false`. The following sequence of steps constitute the counter-example.

Step	**Actor**	**Use case**
1	Administrator ()	Add user (a)
2	Employee (a)	Register
3	Employee (a)	Log in
4	Employee (a)	Place call (a)

Table 2. Counter-example trace

We need to add a pre-condition to the "Place call" use case that will ensure that the IDs of the callee and caller are not the same. We declare a new condition called "Same user IDs" with two parameters for the IDs, and for each valid value in the "Employee ID" type we add an initial "Same user IDs" condition with both parameters set to that value. Next we add the following pre-condition to the "Place call" use case:

Same user IDs (*#self* ID, *#uc* Remote user ID) *is* false

(c) 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`.

(d) A user cannot participate in more than one call at a time. We use the "Absence" pattern to con-

struct 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 Table 3.

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)

Table 3. 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.

6. Conclusions and future work

The main objective of the work presented in this paper was to improve RS by enhancing the currently available processes, techniques and automated tool support for specifying system requirements. 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. Verification of models with SusanX can help developers to identify logical flaws and missing requirements in the models early in the development cycle. Additionally, with Susan developers can get much better insight into their requirements models.

At this stage, Susan has not been applied to any large-scale systems and SusanX still needs to be extensively tested for correctness, 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.

Once sufficiently tested and assessed, Susan and SusanX should be extended to incorporate relationships between use cases and actor generalisation relationships. Furthermore, a user-driven animation feature could be introduced into the SusanX tool to make it even more effective.

References

[1] What Are Your Requirements? A Standish Group Research Note, 2003.

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

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

[4] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 495–499, Trento, Italy, July 1999. Springer.

[5] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property Specification Patterns for Finite-State Verification. In M. Ardis, editor, *Proc. 2nd Workshop on Formal Methods in Software Practice (FMSP-98)*, pages 7–15, New York, 1998. ACM Press.

[6] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-state Verification. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.

[7] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Online Repository for Information about Specificaiton Patterns for Finite-state Verification. Available online: http://patterns.projects.cis.ksu.edu/, Last accessed: February 2004.

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

[9] E. Kindler. Safety and Liveness Properties: A Survey. *Bulletin of the European Association for Theoretical Computer Science*, 53:268–272, 1994.

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

[11] B. Regnell, K. Kimbler, and A. Wesslen. Improving Use Case Driven Approach to Requirements Engineering. In *Proceedings of Second IEEE International Symposium on Requirements Engineering*, March 1995.