Analysis of Structured Use Case Models through Model Checking

Ksenia Ryndina
IBM Zurich Research Laboratory
CH-8803 Rueschlikon
Switzerland
ryn@zurich.ibm.com

Pieter Kritzinger
Department of Computer Science
University of Cape Town
Rondebosch 7701, South Africa
psk@cs.uct.ac.za

Technical Report: CS05-02-00

Abstract

Inadequate requirements specification remains to be one of the predominant causes of software development project failure today. This is mainly due to the lack of suitable processes, techniques and automated tool support available for specifying and analysing system requirements. In this paper we suggest a way to improve the approach to requirements specification that is the most popular at the moment - use case modelling. Despite their popularity, use case models are not adequate for creating comprehensive and precise requirements specifications. We amend the traditional use case metamodel such that more formal and structured models can be built. Further, we define several analysis schemes for these structured use case models that assist in discovering inconsistencies and other errors in the models. These analysis schemes are automated in a tool that we developed called the Structured Use case Model Analyser (SUM Analyser). The SUM Analyser provides an accessible interface that allows the user to construct use case models, configure and execute several analysis options and view the produced results. The existing NuSMV model checker is used to perform the actual verification tasks for the analysis. To facilitate this, the SUM Analyser transforms use case models to NuSMV programs and also interprets the produced results so that they can be understood by the user.

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 [2] 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.

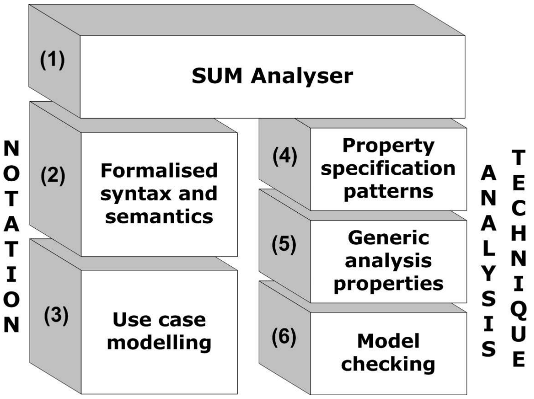
What is it about requirements specification that developers find so challenging? One of the major issues is the lack of adequate processes, techniques and automated tool support available for specifying and analysing system requirements. We thus set out in our research to enhance requirements specification methodology by improving the approach that is the most popular at the moment - *use case modelling* [5, 6]. 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 rigorous analysis of such models impossible. We amend the traditional use case metamodel such that more formal *structured use case models* can be built. Further, we define several analysis schemes for these structured use case models that assist in discovering inconsistencies and other errors in models early in the development cycle. These analysis schemes are automated in a tool that we developed called the *Structured Use case Model Analyser (SUM Analyser)*. The SUM Analyser provides an accessible interface that allows the user to construct use case models, configure and execute several analysis options and view the produced results. The existing NuSMV model checker [8] is used to perform the actual verification tasks for the analysis. To facilitate this, the SUM Analyser transforms use case models to NuSMV programs and also interprets the produced results so that they can be understood by the user.

In order to validate our proposed requirements specification and analysis approach, we performed a case study of a *Cash Management System (CMS)* developed for an international business group. We successfully used the proposed notation to model the CMS requirements and performed various analyses on the models with the SUM Analyser. Numerous errors were identified and remedied during this process and the general state of the requirements specification for the system was considerably improved. To the best of our knowledge, our approach to enhancing use case modelling is unique.

The main objective of this paper is to introduce the structured use case modelling and analysis technique and demonstrate its advantages. The next section provides an overview of the proposed solution. Section 3 gives background to the CMS case study. Section 4 explains the amended use case modelling notation, using examples from the case study to illustrate the various concepts. In Section 5 we show how a structured use case model is mapped to the NuSMV input language for analysis with the NuSMV model checker. The different analysis options offered in the SUM Analyser are discussed in Section 7. Finally, the last two sections describe related work, conclusions and suggestions for future work.

2. Solution Overview

The enhanced technique that we propose uses several existing approaches as building blocks to form an improved solution, as depicted in Figure 1. The notation that we adopt is based on use case modelling [5, 6], shown in block (1) in the diagram. The use case approach to modelling requirements was first presented by Ivar Jacobson [15], but it is now considered to be a part of the *Unified Modeling Language (UML)* [1]. Requirements models in the use case notation are semi-formal and usually consist of diagrams supplemented by text. We extend use case models with a more formal syntax and semantics as shown in block (2), to make them suitable for rigorous automated analysis.



(1)
SUM Analyser
NOTATION
(2)
Formalised syntax and semantics
(3)
Use case modelling
(4)
Property specification patterns
(5)
Generic analysis properties
(6)
Model checking
ANALYSIS
TECHNIQUE

Figure 1. Enhanced Technique for Requirements Analysis

Rigorous analysis of formalised use cases is enabled with *model checking* [9, 18] in our solution, as illustrated in block (6) in Figure 1. Model checking is the process of algorithmically determining whether a behavioural model satisfies certain specification properties, which are usually expressed in some form of temporal logic. Our amendment of the use case notation facilitates creation of high-level behavioural models that capture the desired functionality of a system, and these are then analysed with model checking.

In our research we utilised the NuSMV model checker as the analysis engine for our requirements models. NuSMV is a state-of-the-art *symbolic* model checker, which is based on *Binary Decision Diagrams*. It verifies finite state-transition models expressed in a prescribed NuSMV input language. In order to make use of this tool, we defined a mapping from our structured use case models to NuSMV programs.

Specification properties for NuSMV analysis can be expressed in *Computational Tree Logic (CTL)* [10]. Our solution shelters the developer from the complexities of temporal logic in two ways. First, we define a number of *generic analysis properties* shown in block (5) in Figure 1 that can be used to analyse any structured use case model, which allows the developer to check models without providing any extra input. Second, we make use of *property specification patterns* [11, 12] that allow one to construct simple analysis properties in terms of behavioural patterns and model elements. Specification patterns appear in block (4) in Figure 1 as part of the proposed analysis technique.

The construction, manipulation and analysis of the structured use case models is automated by the SUM Analyser tool, which is represented by block (1) in Figure 1. The SUM Analyser translates use case models to the NuSMV input language for analysis and also interprets the results produced by the model checker in terms of the original use case models.

3. Case Study: A Cash Management System

The case study of the Cash Management System or CMS was made possible through cooperation with an established South African IT company, which we refer to as *SoftCo* in this paper. SoftCo were contracted to develop the CMS for an international business group and at the time of the case study a part of this project was still in progress. The main goal of the CMS is to support management of receipts, as well as coordinate the flow of information between various other computer systems employed by the client company. Examination of the acquired requirements models and documents for the CMS revealed that they were to a large extent ambiguous, inconsistent and incomplete. Our goal was to show that the proposed structured use case notation and analysis schemes offered in the SUM Analyser could improve the quality of this requirements specification.

The requirements specification for the CMS obtained from SoftCo comprised use case diagrams supplemented by informal textual descriptions for each use case. Textual use case descriptions contained information about actors associated with use cases, their main and alternative flows, as well as their pre- and post-conditions. We used a subset of the CMS requirements specification for the purpose of the case study that consisted of 35 use cases, which described the *administration* and *manual handling of receipts* in the system.

Figure 2 shows an extract from one of the revised use case diagrams for the CMS. The diagram depicts three actors: *Administrator*, *Clerk* and *Audit Control System*. The *Administrator* is responsible for adding and deleting valid users within the system. The main user is represented by the *Clerk* actor and needs to *Log In* before gaining access to the system's receipt handling services, such as the saving and printing of receipts. A *Clerk* can have three different roles: *Capture Clerk*, *Enquiry Clerk* and *Supervisor*. Access rights to different services are assigned according to these roles. For instance, a *Capture Clerk* can save receipts but does not have the right to delete them. Only the *Supervisor* has the access right to void receipts. The *include* relationship between the *Print Receipt* and *Post Receipt* use cases indicates that whenever a receipt is printed, it is posted to the accounting and operations systems. Once a receipt is posted, it cannot be deleted from the CMS. A receipt that is saved but not posted can be deleted, however on deletion it is only flagged as deleted thus retaining the information necessary for auditing purposes. After an audit is performed and the information about the deleted receipts is not required anymore, the *Audit Control System* indicates to the CMS that the deleted flags can be cleared. This subset of the CMS requirements and the accompanying use case diagram shown in Figure 2 are used throughout the remainder of the paper for illustrative purposes.

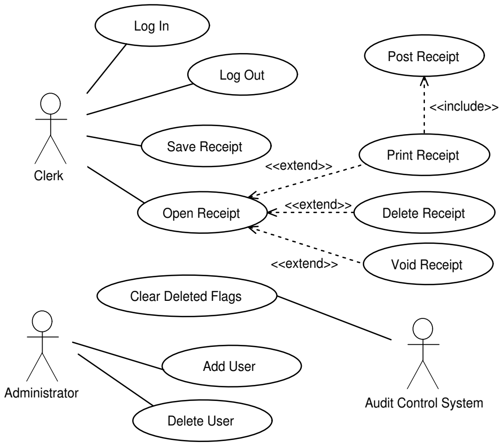


Figure 2. Revised CMS Use Case Diagram

We constructed structured use case models for the CMS requirements in the SUM Analayser. During this process, the provided informal use case descriptions were changed to adhere to our amended use case metamodel and the format prescribed by the SUM Analyser. Several inconsistencies were discovered and remedied during the process of merely structuring and formalising the use case descriptions. Furthermore, the resultant models captured in the SUM Analyser were more precise and comprehensible than the original models.

Numerous errors were identified in the CMS use case models by running them through the analyses in the SUM Analyser. These errors mostly constituted missing use cases, incomplete use case descriptions and logically flawed pre- and post-condition definitions. From the usability perspective, the analysis features offered by the SUM Analyser were found to be very accessible. The feedback provided by the tool in case of discovered errors offered valuable assistance for tracking down sources of the problems.

In the following sections we present the metamodel for structured use case models and our proposed analysis schemes.

4. Structured Use Case Models

We took the fundamental concepts from the standard use case approach and appended them with additional elements to facilitate construction of models suitable for rigorous analysis. The diagram in Figure 3 shows the metamodel for structured use case models.

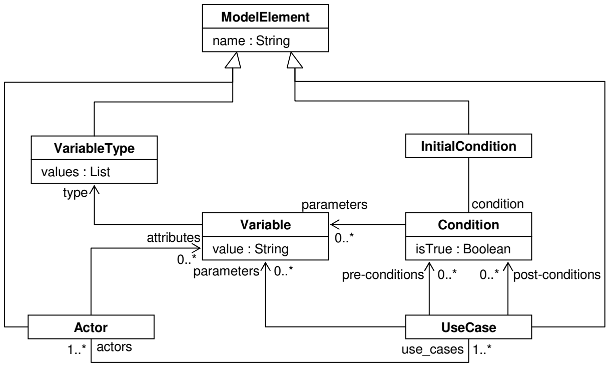
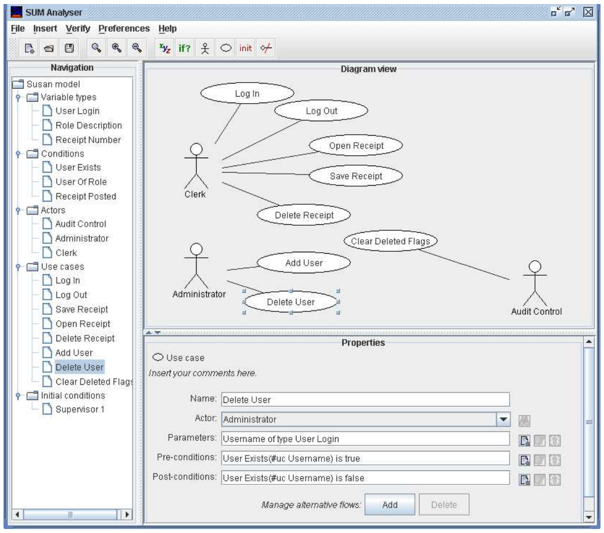


Figure 3. Amended Use Case Metamodel

In accordance with the traditional use case approach, our amended use case notation captures requirements in terms of actors as external entities interacting with the system and use cases representing the required system services. Furthermore, we formalise the existing notion of use case pre- and post-conditions by defining their structure and relation to use cases precisely in the metamodel shown in Figure 3. A use case model in our approach is a high-level behavioural model that represents the required interaction between actors and the system through use case *activation*. *Initial conditions* are used to describe the system state before any interaction between actors and the system occurs, thereafter use case activations alter the system state. When an actor activates a use case, the pre-conditions of that use case are queried against the system state and if they hold then the activation is said to be successful. On a successful use case activation, post-conditions of the use case are used to adjust the current system state.

A structured use case model consists of a use case diagram showing the graphical representation of actors, use cases and their associations. A predefined format is used to define additional properties for actors and use cases, such as *actor attributes* and *use case parameters*. Attributes can be defined to capture an actor's particulars that the system needs to access in order to deliver services to that actor. Use case parameters state the information that an actor needs to pass to the system when activating that use case. Furthermore, initial conditions, variable types and condition definitions need to be specified in order to complete a structured use case model. This can be seen in Figure 4 that shows a screenshot of the main model editor window in the SUM Analyser. Separate windows are available in the tool for the different analyses of models.



Insert your comments here.

Name: Delete User
Actor: Administrator
Parameters: Username of type User Login
Pre-conditions: User Exists(#uc Username) is true
Post-conditions: User Exists(#uc Username) is false

Figure 4. Screenshot of SUM Analyser

From the metamodel in Figure 3 it can be seen that relationships among use cases such as *extend* and *include*, or actor generalisation relationships are not supported. In its current state our technique is built around the fundamental features of use case models only, as our goal was to test the approach first before incorporating the additional use case modelling features. However, in [19] we discuss some preliminary and potentially extensible solutions for expressing use case relationships and actor hierarchies in structured use case models. These solutions allowed us to "flatten" the CMS use case diagrams by taking out the unsupported relations between actors and use cases, but at the same time retaining most of the semantics of these relations by including additional textual elements.

We next discuss a few examples of model element definitions extracted from the CMS use case models constructed in the SUM Analyser. Below are the definitions for *Delete User* and *Void Receipt* use cases and the *Clerk* actor, all shown in Figure 2. Additionally, one initial condition and further elements from the structured use case model are also shown below.

USE CASE 1
name: *Delete User*
actors: *Administrator*
parameters: *Username of type User Login*
pre-conditions: *User Exists (#uc Username) is true*
post-conditions: *User Exists (#uc Username) is false*

USE CASE 2
name: *Void Receipt*
actors: *Clerk*
parameters: *Receipt of type Receipt Number*
pre-conditions: *Logged In (#self Username) is true, User Of Role (#self Username, #Supervisor) is true, Receipt Opened (#uc Receipt) is true, Receipt Posted (#uc Receipt) is true*
post-conditions: *Receipt Reversed (#uc Receipt) is true*

VARIABLE TYPE 1
name: *User Login*
values: *jbloggs, mjane, agatonye*

VARIABLE TYPE 2
name: *Role Description*
values: *Capture Clerk, Enquiry Clerk, Supervisor*

ACTOR 1
name: *Clerk*
attributes: *Username of type User Login*

CONDITION 1
name: *User Exists*
parameters: *Username of type User Login*

CONDITION 2
name: *User Of Role*
parameters: *Username of type User Login, Role of type Role Description*

INITIAL CONDITION 1
name: *Supervisor 1*
condition: *User Of Role (#jbloggs, #Supervisor)*

The definition of the *Delete User* use case (USE CASE 1) is quite straightforward. The definition indicates that the *Administrator* is the only actor that can activate this use case and that the *Username* of the user to be deleted needs to be provided to the system as a use case parameter. Note that each use case parameter has an associated variable type, where each variable type defines a finite set of symbolic values. In this case, *Username* is of type *User Login* (VARIABLE TYPE 1) and hence can take on any of the three valid symbolic values: *jbloggs*, *mjane* and *agatonye*. The pre-condition for this use case states that an activation is successful if the user with the provided *Username* exists at the time of activation. As indicated by the post-condition, on successful activation the state of the system changes to reflect that this user no longer exists. Note that each use case pre- and post-condition corresponds to a condition declaration within the model, where the number and type of condition parameters are defined. In this example, the *User Exists* condition is declared in the CONDITION 1 definition. This definition indicates that the condition has one parameter of variable type *User Login*. The *#uc* prefix in *User Exists (#uc Username) is true* indicates that at the time of activation, the value of the use case parameter *Username* should be used for the evaluation of this pre-condition.

The definition of the *Void Receipt* use case (USE CASE 2) states that for a successful activation the *Clerk* must be logged in, the *Clerk* must have *Supervisor* access rights, and the receipt under consideration must be opened and posted. If these pre-conditions are satisfied at the time of the use case activation, then the receipt is reversed in the accounting and operations system. The *#self* prefix in *Logged In (#self Username) is true* indicates that the *Username* attribute of the *Clerk* actor must be used to evaluate this pre-condition. Two more options for pre- and post-condition parameters besides *#uc* and *#self* are available. One is illustrated in *User Of Role (#self Username, #Supervisor) is true*, where a literal value *Supervisor* from the *Role Description* type (VARIABLE TYPE 2) is used. This pre-condition checks that the *Clerk*'s *Username* is associated with the *Supervisor* role. The last option *#forall* allows to check that a condition holds for all values of a particular variable type. For instance, we can check that nobody is logged in with *Logged In (#forall User Login) is false*.

Both use cases in the above example have only one flow and thus one set of pre- and post-conditions. If a use case has alternative flows, a pre- and post-condition set for each flow is included in the use case definition. All the pre- and post-conditions in the same set are implicitly joined with an AND logical operator, while pre- and post-condition sets are implicitly joined with an OR.

The initial condition definition in the above example (INITIAL CONDITION 1) states that the *Clerk* with *Username jbloggs* is assigned a *Supervisor* role in the initial state of the system.

A structured use case model created in the SUM Analyser is translated into the NuSMV input language and then all the possible behaviours are checked with the NuSMV model checker. With this in mind, the concept of condition parameters is comparable to *formal* and *actual* parameters of methods in programming languages like Java. In a structured use case model, a condition declaration (such as CONDITION 1) 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 (such as USE CASE 1), the user assigns each of the formal parameters to an actual parameter as described before. During the verification 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 literal 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. In contrast with a condition definition, we say that a *condition instance* has its parameters assigned to literal values. *User Exists (jbloggs)* is an example of a condition instance. A use case with values assigned to its parameters and the attributes of its associated actor is called a *use case instance*. A use case instance corresponds to a use case activation, such as *Administrator.Delete User (jbloggs)*. The mapping from a structured use case model to the NuSMV input language is described next.

5. Mapping Use Case Models to NuSMV

This section describes the mapping of structured use case models to the NuSMV input language. For an explanation of the NuSMV input language itself, we refer the reader to [8] and the documentation for the NuSMV model checker.

A structured use case model as described in the previous section can be seen as a finite state machine, where values of condition instances define the system state and state transitions are defined by activation of use case instances. In our mapping to NuSMV, we represent each condition instance in a use case model as a *state variable*. These condition variables are initialised in accordance to the initial conditions in the model. For each use case instance, a NuSMV *module* is defined inside which the condition variables are reassigned values as indicated by the pre- and post-conditions of the use case. The following extract from a NuSMV program shows the modules generated for instances of the *Delete User* and *Void Receipt* use cases discussed in the previous section.

```
MODULE DeleteUser$0$(UserExists$0$)
VAR
  return : boolean;
ASSIGN
  init(return) := 0;
  next(return) :=
  case
    (UserExists$0$ : 1;
    1 : 0;
  esac;
  next(UserExists$0$) :=
  case
    (UserExists$0$) : 0;
    1 : UserExists$0$;
  esac;
FAIRNESS running;

MODULE VoidReceipt$0$0$(LoggedIn$0$, UserOfRole$0$2$,
 ReceiptOpened$0$, ReceiptPosted$0$, ReceiptReversed$0$)
VAR
```

```
21   return : boolean;
22 ASSIGN
23   init(return) := 0;
24   next(return) :=
25   case
26     (LoggedIn$0$ & UserOfRole$0$1$ & ReceiptOpened$0$
27                  & ReceiptPosted$0$ : 1;
28     1 : 0;
29   esac;
30   next(ReceiptReversed$0$) :=
31   case
32     (LoggedIn$0$ & UserOfRole$0$2$ & ReceiptOpened$0$
33                  & ReceiptPosted$0$ : 1;
34     1 : ReceiptReversed$0$;
35   esac;
36 FAIRNESS running;
```

A special scheme is used to generate compact and unique names for condition variables and use case instance modules in a NuSMV program. During the name generation process, spaces are taken out from use case and condition names and their parameter values are replaced by numbers. For example, the `DeleteUser$0$` use case module is used to represent the *Administrator.Delete User (jbloggs)* use case instance.

Inside a use case instance module, the pre-conditions of the use case instance are checked. This is done by considering the values of the corresponding condition variables. Passing the appropriate condition variables to each use case instance module as parameters provides the modules access to the values of these variables. Additionally, condition variables for the post-conditions of a use case instance also need to be passed to its module as they get re-assigned there (lines 12-16, 34-39). In the example above, the passing of parameters into the use case modules is shown in lines 1, 21-22.

As can be seen in lines 3 and 24 above, a boolean variable called `return` is declared inside each use case instance module. This variable is used to determine whether a use case activation represented by the use case instance module is successful or not. This variable is first initialised to 0 (lines 6, 27) and if the pre-conditions of the use case are met then its value is re-assigned to 1 (lines 7-11, 28-33).

There is also one main module in every NuSMV program, where we place condition variable declarations and initialisations. Each use case instance module is instantiated as a *process* in the main module. Using processes in NuSMV and including the FAIRNESS clause in the use case modules (lines 18-19, 41-42), ensures that during verification these modules are instantiated nondeterministically. Each instantiation represents a use case instance activation. Nondeterministic choice between activations allows us to check all the possible ways in which the system can be used.

Finally, the NuSMV program needs logic specification properties to perform verification. CTL specifications for verification are included in the main module of a NuSMV program. More details on how these specifications are generated is given in the following section.

6. Analysis of Models with the SUM Analyser

The SUM Analyser supports two modes of analysis or verification: generic and model-specific. An overview of how verification is performed with the SUM Analyser tool and NuSMV is given in Figure 5. The mappings from structured use case models to NuSMV described in the previous section are used to translate the models created in the SUM Analayser to NuSMV programs. Generic verification can be applied to any use case model and the CTL properties for this verification mode are embedded into the SUM Analyser (see Appendix). They are simply parameterised for the current model and passed to the NuSMV model checker as shown in the diagram. The SUM Analyser provides a number of specification patterns that assist the user in constructing model-specific properties for verification. As can be seen, these are automatically translated to CTL by the SUM Analayser. Finally, verification results are interpreted for the user in terms of the original use case model. The details of the two verification modes are described next.

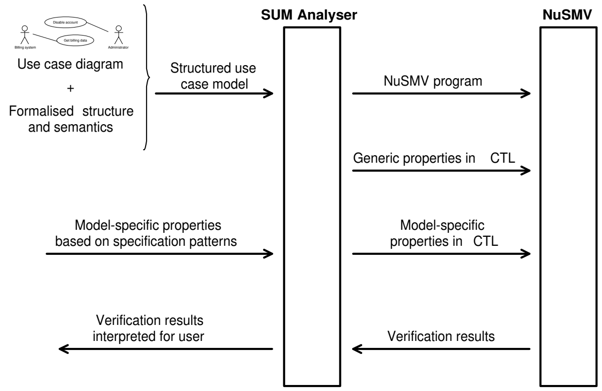

SUM Analyser
NuSMV
Disable account
Get billing data
Billing system
Administrator
Use case diagram
+
Formalised structure and semantics
Structured use case model
NuSMV program
Generic properties in CTL
Model-specific properties based on specification patterns
Model-specific properties in CTL
Verification results interpreted for user
Verification results

Figure 5. Verification with SUM Analyser

6.1. Generic Verification

Generic verification of a structured use case model in the SUM Analyser does not require any additional input from the user. This verification mode is used to analyse use cases for *liveness* and conditions for *reversibility*.

Liveness of use cases: An informal definition of the liveness property is that "something good will always eventually happen" [16]. We define three liveness categories for a use case: *Dead*, *Transient* and *Live*. The SUM Analayser checks a model and places each use case instance into one of these categories.

(a) Dead: Successful activation of the use case instance is not possible. Usually, one should be alarmed if all instances of a use case fall into the *Dead* category, because a use case that can never be successfully activated serves no purpose in a 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.

(c) Live: It is possible to activate the use case instance infinitely many times. Most use case instances in a model usually fall into this category.

Reversibility of conditions: The SUM Analayser checks 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 infinitely many times. Most conditions fall into this category.

Verification for liveness of use cases and reversibility of conditions with the SUM Analayser 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 him of potential errors in the model.

During liveness analysis of the use cases from our CMS case study, we discovered that all instances of the *Open Receipt* use case were *Live*. This was in accordance with our expectations since any *Clerk* can open a receipt an unlimited number of times. Furthermore, all instances of the *Void Receipt* use case were also reported *Live*. This meant that a particular receipt could be voided more than once. Since every time a receipt is voided the corresponding transaction is reversed in the accounting and operations systems, this situation would ultimately result in incorrect transaction records. Taking into consideration that these transactions could involve very large amounts of money, such a flaw in the requirements model could have devastating consequences. The model was corrected by adding a precondition to the *Void Receipt* use case that ensured that the receipt in question had not been voided before.

Verifying the CMS conditions for reversibility revealed that all the instances of the *Receipt Saved* condition were *Irreversible*. At a closer inspection, we discovered that according to the requirements model when a receipt was deleted it was just marked with a deleted flag and still considered to be “saved” within the system. This also meant that a deleted receipt could be opened as any other saved receipt, which was not desirable. We remedied this situation by adding the following post-condition to the *Delete Receipt* use case: *Receipt Saved (#uc Receipt) is false*.

Several other errors were discovered and corrected in the structured use case models for the CMS during generic verification with the SUM Analayser. Careful inspection of the verification results and a good knowledge of the liveness and reversibility categories were necessary during this process.

6.2. Model-Specific Verification

Verification against generic properties yields useful results, but because the generic properties cannot be used to test model-specific behaviour, this type of analysis is limited. We present the user with property specification patterns for the creation of custom properties. 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 are generalised descriptions of commonly-sought behaviours for verification of finite state systems. Specification patterns were first proposed by Dwyer *et al* in [11] and further supported by empirical studies [12]. Dwyer *et al* developed a system of specification patterns, which comprises a set of patterns that are organised into a hierarchy showing the relationships between them. We tailored the original pattern hierarchy slightly to suit our specific needs for use case model analysis. In our augmented pattern hierarchy we did not include the patterns that were rarely used as shown by the surveys in [12], furthermore we added several new patterns to it. The SUM Analayser pattern hierarchy is shown in Figure 6. The original patterns that were not included in our hierarchy are indicated in grey and the new patterns are shown with a border.

Instantiation of patterns to construct behavioural properties is performed as follows. 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. For our use case models, pattern variables can be constructed from: condition instances and the logical operators NOT (!), AND (&), OR (|) and implication (→).

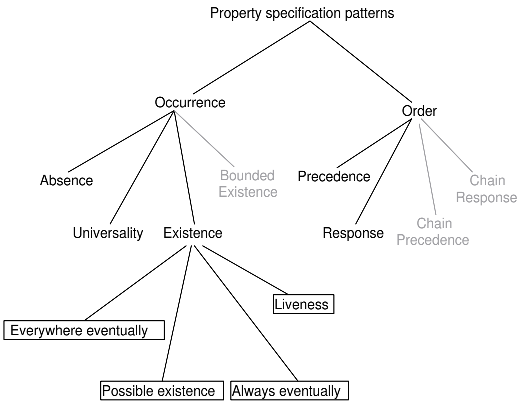


Figure 6. Property Specification Pattern Hierarchy for SUM Analyser

Once the user chooses a pattern and fills in the pattern variables, the corresponding CTL formula can be generated.

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

(a) Absence: *Safety properties* can be constructed using this pattern. An informal definition of a safety property is that "something bad will never happen" [16].

(b) Universality: 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: 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.

In the SUM Analayser, we used the mappings to CTL as defined by Dwyer *et al* for all the patterns except the new *Existence* sub-patterns, for which we defined our own mappings (see Appendix).

The NuSMV model checker generates a counter-example trace whenever the verified property is found to be false. During model-specific verification, such traces are interpreted for the user to demonstrate what use case activations lead to violation of the verified property. However, certain properties such as *Possible existence* do not generate counter-examples.

We used model-specific verification in the SUM Analyser to verify that the CMS use case models satisfied certain constraints and also discovered several further flaws in the models. For instance, using the *Universality* pattern we verified that once a receipt is posted it cannot be deleted in the system. The property that was constructed in the SUM Analyser to check this is *Universality of (Flagged Deleted (a) → ! Receipt Posted (a))*. During verification, *a* is replaced by all possible values from the *Receipt Number* variable type.

Using the *Absence* pattern, we constructed a property to check that only valid users can log into the system: *Absence of (Logged In (b) & ! User Exists (b))*. During model checking, *b* is replaced with all possible values from the *User Login* variable type. This property was evaluated to false in the model with the following counter-example:

1. Administrator.Add User (jbloggs) - successful
2. Clerk (jbloggs).Log In () - successful
3. Administrator.Delete User (jbloggs) - successful

The counter-example shows that the *Administrator* can successfully delete the user *jbloggs* while a *Clerk* with this *Username* is logged into the system. This flaw was remedied by adding a pre-condition to the *Delete User* use case to ensure that the currently logged in users cannot be deleted.

The *Receipt Posted* condition was analysed using the *Existence* patterns in the SUM Analyser. We used the *Possible Existence* pattern to determine that it is possible for receipts to be posted successfully within the system as required. However, we also discovered that instances of the *Receipt Posted* condition are not *Always Eventually* true. This result was also plausible since those receipts that are deleted can never be posted in the system.

The model-specific verification mode of the SUM Analyser allowed us to perform valuable analyses of the CMS use case models. A grasp of the patterns and a basic understanding of the logical operators were required during construction of model-specific analysis properties. On the other hand, counter-examples were very easy to understand and proved valuable in resolving why a verification property failed.

It is well-known that the main drawback of model checking is its performance, in other words the time it takes to compute verification results. Since the model checking algorithm performs an exhaustive search of all the possible execution paths of a given model, verification time increases exponentially with the size of the model. The NuSMV model checker that we chose for this work performed relatively well in obtaining the analysis results for the CMS use case models. All verification results for the CMS use case models could be obtained within a period of 4 to 1300 seconds. However, some large models had to be separated into smaller models using appropriate abstraction techniques to ensure that verification results remained valid for the entire model. More details about the performance of the SUM Analyser can be found in [19].

7. Related Work

Many different techniques for specifying system requirements have been proposed by researches, but very few have gained acceptance in the industry. On the one hand, there are techniques based on expression of requirements in natural language such as ARM [21] and SREM [20]. Natural language requirements specifications are easy to produce and can be understood by all system stakeholders. However, such specifications can be very ambiguous and the possibility of performing rigorous analysis on them is extremely limited. On the other hand, there are approaches that propose formal requirements notations suitable for automated analysis; these include SCR [14] and PAISLey [4]. These formal approaches can be very effective when applied to real-time embedded or safety-critical systems, however in other domains developers seem reluctant to use such complicated techniques. The middle-ground between informal requirements in natural language and formalised requirements models is taken up by semi-formal graphical representations of system requirements, which emphasise the importance of visualising requirements models. UML use case, sequence and state diagrams fall into this category. Owing to its flexibility, UML has become the *de facto* standard in modelling software systems, with requirements specification being done predominantly with use case diagrams.

Several attempts have been made to address the drawbacks of use case modelling. Hausmann *et al* [13] propose refining use cases with UML activity diagrams and expressing their pre- and post-conditions in terms of UML collaboration diagrams. This approach allows for static analysis of conflicts and dependencies in use case models. Back *et al* [3] formalise use cases with *contracts* defined in refinement calculus, which facilitates rigorous analysis of use case models for properties such as “achievability” and safety. The complexity of the mathematical notation underlying this approach and the absence of tool support automating the analysis makes this technique impractical. Our proposed solution enhances use case modelling by formalising the models and facilitating their automated analysis while keeping the intricacies of the analysis hidden from the user.

The use of model checking has proved to be very successful in verifying hardware designs, and recently its application to software models has notably increased [7]. McUmber *et al* [17] have developed a framework and a number of tools for translating UML class and state diagrams to formal specifications that can be simulated and analysed by model checkers. In a similar way, we use the NuSMV model checker as a verification engine of high-level behavioural models created in the SUM Analyser.

8. Conclusion

The main objective of the work presented in this paper was to provide better support for requirements specification and analysis. We did this by developing an enhanced technique based on use case modelling and the supporting SUM Analyser tool that uses the NuSMV model checker for verification. Our approach allows for the creation of structured use case models that are more complete, consistent and correct. Verification of models with the SUM Analyser can help developers to identify logical flaws and missing requirements in the models early in the development cycle. Additionally, by using the SUM Analyser developers can get much better insight into their requirements models. The work was successfully validated with the Cash Management System case study.

A number of further developments of the approach and the SUM Analayser tool would be interesting and beneficial. These include extension of the amended use case metamodel to incorporate use case and actor relationships, adding new features to the SUM Analayser tool such as use case animation and undertaking further case studies.

Acknowledgement: The authors thank Jana Koehler and Jochen M. Küster for their valuable comments on an earlier version of this paper.

References

[1] UML 2.0 Superstructure Final Adopted Specification, ptc/03-08-02. OMG Document, 2003.

[2] What Are Your Requirements? The Standish Group International, 2003.

[3] R.-J. Back, L. Petre, and I. Porres-Paltor. Analyzing UML Use Cases as Contracts. In *UML'99 - Second International Conference on the Unified Modeling Language: Beyond the Standard*, pages 518 – 533. Springer-Verlag, October 1999.

[4] E. Berliner and P. Zave. An Experiment in Technology Transfer: PAISLey Specification of Requirements for an Undersea Lightwave Cable System. In *Proc 9th International Conference on Software Engineering*, pages 42–50, Monterey, California, United States, 1987. IEEE Computer Society Press.

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

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

[7] W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. Reese. Model Checking Large Software Specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

[8] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc International Conference on Computer-Aided Verification*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[9] E. Clarke, E. Emerson, and A. Sisla. Automatic Verification of Finite State Concurrent Systems using Temporal Logic. In *ACM Trans on Programming Languages and Systems*, volume 8, pages 244–263, 1986.

[10] E. M. Clarke and E. A. Emerson. Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. In *Logics of Programs: Workshop*, volume 131 of LNCS, Yorktown Heights, New York, May 1981. Springer.

[11] 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*, pages 7–15, New York, 1998. ACM Press.

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

[13] J. H. Hausmann, R. Heckel, and G. Taentzer. Detection of Conflicting Functional Requirements in a Use Case-Driven Approach: A Static Analysis Technique based on Graph Transformation. In *Proc 24th International Conference on Software Engineering*, pages 105–115, Orlando, Florida, 2002. ACM Press.

[14] C. Heitmeyer, J. Kirby, and B. Labaw. The SCR Method for Formally Specifying, Verifying, and Validating Requirements: Tool Support. In *Proc 19th International Conference on Software Engineering*, pages 610–611, Boston, Massachusetts, United States, 1997. ACM Press.

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

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

[17] W. E. McUmber and B. H. C. Cheng. A General Framework for Formalizing UML with Formal Languages. In *Proc 23rd International Conference on Software Engineering*, pages 433–442, Toronto, Ontario, Canada, 2001. IEEE Computer Society.

[18] J. P. Quielle and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proc 5th International Symposium on Programming. LNCS 137*, pages 337–350, New York, 1981. Springer.

[19] K. Ryndina. Improving Requirements Engineering: An Enhanced Requirements Modelling and Analysis Method. Master's thesis, Department of Computer Science at University of Cape Town, South Africa, November 2004.

[20] P. Scheffer, W. Rzepka, and I. A.H. Stone. A Large System Evaluation of SREM. In *Proc 7th International Conference on Software Engineering*, pages 172–180, Orlando, Florida, United States, 1984.

[21] W. M. Wilson, L. H. Rosenberg, and L. E. Hyatt. Automated Analysis of Requirement Specifications. In *Proc 19th International Conference on Software Engineering*, pages 161–171, Boston, Massachusetts, United States, 1997. ACM Press.

APPENDIX

The table below shows the CTL formulae that are used to determine liveness categories for use case instances and reversibility categories for condition instances in the SUM Analyser. In the table, u stands for the name of a NuSMV use case instance process such as `activated_DeleteUser$0$` for example. Similarly, c stands for the name of a NuSMV condition variable such as `UserExists$0$`. The table also shows the mappings of the *Existence* patterns to CTL that we defined. In the table, v stands for a pattern variable composed of condition instances and logical operators as explained in Section 6.2.

Liveness category	**CTL formula**
Dead	!EF u
Transient	EF u & !AG EF u
Live	AG EF u
Reversibility category	**CTL formula**
Constant	!EF c
Irreversible	EF c & AG ($c \rightarrow$ AG c)
Finitely-reversible	EF c & !AG EF c
Reversible	EF c & AG ($c \rightarrow$ EF c)
Existence pattern	**CTL formula**
Everywhere eventually	AF v
Possible existence	EF v
Always eventually	AG AF v
Liveness	AG EF v