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

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 structured models with a precise meaning 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

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. We thus set out in our research to enhance requirements specification methodology by improving one of the most popular approaches at the moment - *use case modelling* [6, 7].

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 *structured* use case models with a precise meaning 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.

The main contribution of our work is allowing the developer to perform rigorous analyses of use case models, without the need to understand the complexities underlying the model formalisation and analysis. While using some existing techniques and tools, our approach is novel from a number of perspectives. Firstly, we propose original amendments to the traditional use case models, such that they represent high-level behavioural system models (Section 4). Secondly, we successfully extend the use of the powerful model checking techniques to a domain where it has not

been applied before - requirements analysis. Thirdly, we define generic analysis properties for use case models that can be checked by the developer with a "push of a button" (Section 6.1). Lastly, we employ specification patterns in construction of analysis properties for use case models and thus examine another way of reducing the obstacles faced by the developer when using formal analysis techniques (Section 6.2).

The objective of this paper is to introduce the proposed 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 6. Finally, the last three sections respectively describe evaluation of the case study, 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 [6, 7, 2], shown in block (3) in the diagram. The use case approach to modelling requirements was first presented by Ivar Jacobson [16], but it is now considered to be a part of the *Unified Modeling Language (UML)* [7, 2]. Requirements models in the use case notation are informal and consist of diagrams supplemented by text. The textual descriptions supplementing use case diagrams are usually written in natural language and comprise details such as use case flows, priority, trigger events, preconditions and post-conditions. Use case flows describe the possible scenarios of interaction between the system and its environment during the use case delivery. Typically, a use case has one main flow and a number of alternative flows. In traditional use case modelling, the developer is free to decide what information to add or omit from the supplementary use case descriptions.

We extend use case models with structured syntax and precise semantics as shown in block (2), to make them suitable for rigorous automated analysis. We call the amended use case modelling notation "structured" use cases, as it supports the creation of use case models comprising well-defined parts expressed in structured text instead of natural language.

Rigorous analysis of structured use cases is enabled with *model checking* [9, 19] in our solution, as illustrated in block

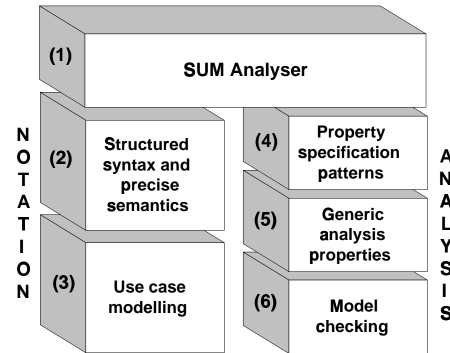

Figure 1. Enhanced Use Case Modelling and Analysis Approach

(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 Background

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 use case diagrams for the CMS. The use cases depicted in this diagram are used throughout the remainder of the paper for illustrative purposes.

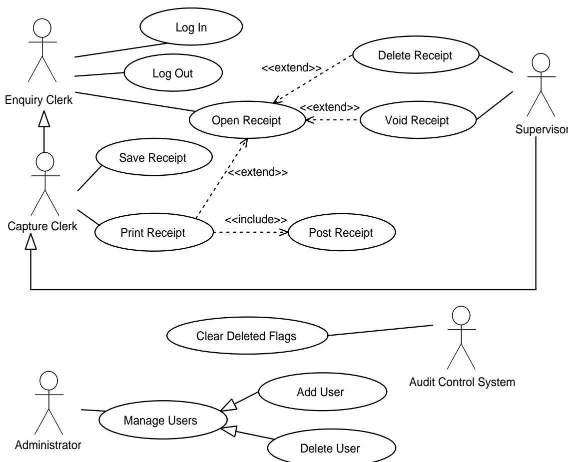

Figure 2. CMS Use Case Diagram

The diagram in Figure 2 depicts five actors. The *Administrator* actor is responsible for managing users within the system, by adding and deleting valid users. *Enquiry Clerk*, *Capture Clerk* and *Supervisor* represent the main users of the system. Any of these three clerk actors needs to *Log In* before gaining access to the system's receipt handling services, such as the saving and printing of receipts. These actors are related with an actor hierarchy relationship, capturing their different access rights. 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.

4. Structured Use Case Models

In this section we present the metamodel for structured use case models and illustrate the creation of a structured use case model with the running example from the CMS case study. Instead of presenting the rules for the structured textual syntax used to capture various model elements, we demonstrate the syntax using concrete examples.

Before introducing the metamodel, we first present the general view on modelling system behaviour requirements that is assumed in our approach. In agreement with the standard use case modelling, the system under consideration is treated as a "black box". The diagram in Figure 3 illustrates the perspective on actor-system interaction taken by the structured use case approach.

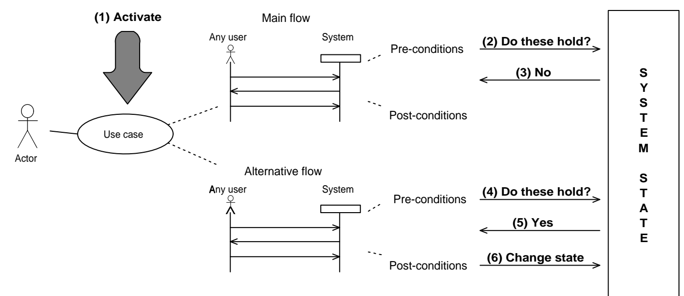

Figure 3. Interaction Between Actor and System in Proposed Approach

The actor can call upon the system's services by *activating* use cases. The system itself is described by the *system state*, which is dynamic and changes in time as a result of use case activations. Predicates referred to as *conditions* in the structured use case approach are used to collectively

represent the state of the system, where at any time the system can be queried for the value of any one of these conditions. For example, one of the conditions describing the state of the CMS system introduced could specify whether a particular receipt is saved within the system.

As shown in Figure 3, each use case is associated with a number of flows and each flow has pre- and post-conditions. The diagram shows what happens during a use case activation with six numbered steps. After a use case is activated by an actor (1), the pre-conditions of its main flow are queried against the current system state (2). Suppose that these pre-conditions do not hold (3), then the alternative flow of the use case is considered. Pre-conditions of the alternative flow are queried (3) and this time they are satisfied in the state of the system (4). Since the pre-conditions are satisfied, the post-conditions of that flow are used to change the system state (5). When pre-conditions for one of the flows hold, the use case activation is said to be *successful*.

As explained above, use case models become more dynamic in the proposed approach than in the standard use case modelling approach. This new view of system requirements modelling allows us to incorporate verification with model checking that explores all the possible interactions between the actors and the system for a particular model.

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. Note that use case models were one of the few parts of UML that were not affected by major changes during the shift from UML 1.x to UML 2.0. Therefore, our proposed approach can be seen as an enhancement of use case modelling as it is defined in UML 1.x or UML 2.0. The diagram in Figure 4 shows the metamodel for structured use case models.

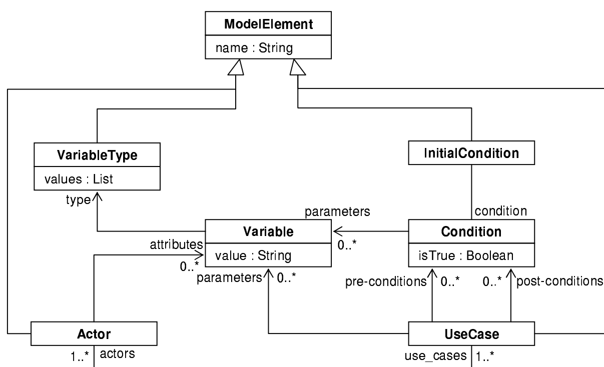

Figure 4. Structured Use Case Metamodel

The aggregation relationships in Figure 4 show that a structured use case model comprises four different types of elements: actors, use cases, conditions and variable types. For each of these modelling elements the metamodel prescribes a number of *properties* that capture information related to that element.

A structured use case model consists of a use case diagram showing the graphical representation of actors, use cases and their associations. For each actor and use case in the diagram, textual properties are additionally defined. Conditions and variable types do not have graphical representations; these elements are completely textual. Each element appearing in the metamodel shown in Figure 4 is explained next.

Two properties are defined for the **Actor** element in the metamodel: a name and a list of **attributes**. Attributes describe an actor's particulars that the system needs to access in order to deliver services represented by use cases to that actor. For example, an *Enquiry Clerk* has an attribute called *Username* that he needs to provide to the CMS system in order to log in. Each actor attribute is regarded as a **Variable**, and each variable has an associated **type** as shown in the metamodel. Each **VariableType** is associated with a finite number of symbolic **values**, which are essentially string literals that can only be compared for equivalence. Two symbolic variables are equal if their values are set to identical string literals.

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 condition: a name, a **parameter** list and a truth-value (**isTrue**). **InitialConditions** are used to describe the system state before any interaction between actors and the system occurs.

A **UseCase** has five properties: a name, its associated actors, 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 literal value for each of its parameters is passed to the system.

From the metamodel in Figure 4 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, we propose a potentially extensible solution for expressing use case relationships and actor hierarchies in structured use case models given the current metamodel. This solution was applied to the CMS case study, where the provided use case model had to be "flattened" before analysing them in the SUM Analyser.

4.1 Flattening of Use Case Models

In our running CMS example, the use case diagram in Figure 2 is flattened to produce the diagram in Figure 5. We next describe how to achieve flattening of a use case

model by eliminating each of the four possible relationships between actors and use cases.

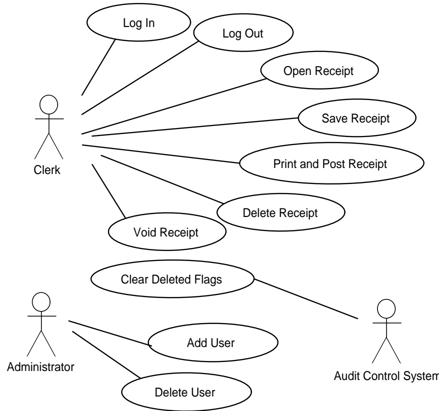

Figure 5. Flattened CMS Use Case Diagram

Use case generalisation: In such a relationship, the general use case is abstract while the concrete behaviour of the system is captured by the use case specialising it. For instance, consider the general use case *Manage users* and the two use cases specialising it in Figure 2. This relationship adds structure to the use case diagram, but the model without it still represents the same behaviour. When creating a flattened use case model from a standard use case diagram, only specialised use cases are included.

Use case include: Included use cases are eliminated in flattened models and hence one cannot show shared behaviour between use cases. In the CMS example, the *Post Receipt* use case is removed from the use case model while the *Print Receipt* remains and is renamed to *Print and Post Receipt*.

Use case extend: When two use cases are joined with the extend relationship, they are both included in the flattened model. The semantics of the relationship are preserved through declaring pre- and post-conditions on these use cases that state that the extending use case can only be activated after the extended one. For example, in the CMS system the *Void Receipt* use case extends the *Open Receipt* use case. For the flattened model, we remove the extends relationship and ensure that a post-condition of the main flow through the *Open Receipt* use case states that a receipt has been opened and make this also the pre-condition for the *Void Receipt* use case flows. It is also possible that the extension point for the extend relationship appears somewhere in the middle of a use case rather then at the end. In this case, the extended use case has to be split into two use cases in the flattened model and then once again pre- and post-conditions can be used to capture the semantics of the relationship.

Actor generalisation: When actor generalisation is used in a use case model, only the general actor is carried through into the flattened model. A condition is then defined in the model that has the identifying actor attributes as parameters and can be used to distinguish which of the specialised actors a particular general actor instance represents. Each of the use cases connected with the specialised actors get associated with the general actor, but a pre-condition that checks the identity of the actor is added to each of these use cases. For our CMS example, we retain the *Enquiry Clerk* actor and give it a more general name, *Clerk*. The *Capture Clerk* and *Supervisor* actors are removed from the model and all the use cases previously associated with them get associated with the *Clerk* actor. This can be seen in the flattened use case diagram in Figure 5. We add pre-conditions to these use cases that check the role of the actor on use case activation to ensure that access control is preserved.

4.2 Example and Detailed Description

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 5. Additionally, one initial condition and some further elements from the structured use case model are also shown below. Textual descriptions for the complete structured use case model are not included due to space restrictions.

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. Generating NuSMV Programs from Use Case Models

This section describes the mapping of a structured use case model to the internal representation of a system in a model checker. We demonstrate how a program in the NuSMV input language is generated from a structured use case model.

In all model checkers, systems are viewed as *Kripke structures* [18]. A Kripke structure is essentially a nonde-

terministic finite state machine, where the states are labelled with propositions that hold in that state. A proposition is simply a statement that can either be true or false. For verification with NuSMV, a structured use case model is used to generate a NuSMV program that describes a Kripke structure. For this, a structured use case model is regarded 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. Initial conditions defined in a structured use case model are used to define the initial state of a Kripke structure.

Figure 6 shows a subset of states and transitions from a Kripke structure representing the CMS structured use case model during verification.

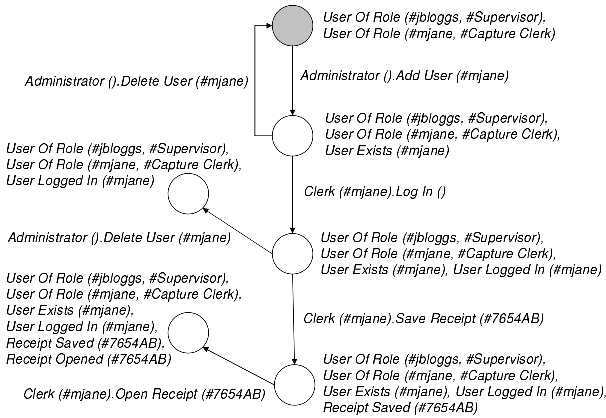

Figure 6. Example of a Kripke Structure

During verification by model checking with NuSMV, all possible paths through a Kripke structure defined by a NuSMV program are explored. In the diagram in Figure 6, several execution paths consisting of use case activations are represented all starting in the initial state marked in grey. For example, one trace is: *Administrator ().Add User (#mjane)*, *Clerk (#mjane).Log In ()*, *Clerk (#mjane).Save Receipt (#7654AB)*. In this trace, a user *mjane* is first added to the system by the administrator, *mjane* logs in and saves a new receipt. At each step in the execution path, the changes to the system state can be observed. Another execution trace is *Administrator ().Add User (#mjane)*, *Administrator ().Delete User (#mjane)*. This trace shows that it is possible to arrive back at the initial state of the system.

The SUM Analyser allows the user to create a structured use case model with the aid of a graphical editor. The internal representation of use case models in the SUM Analyser is based on the metamodel presented in Section 4. At runtime, the SUM Analyser processes a structured use case model as a collection of objects in memory and these objects are serialised when persistence is required. During the generation of the NuSMV input program, the use case model is in memory and as its elements are traversed, lines of NuSMV code are generated accordingly. This straightforward method of creating NuSMV programs was deemed sufficient for our purpose. However, if we intended to analyse the mapping between structured use case models and Kripke structures specified in NuSMV programs further, we would need to define the mapping as a model transformation or adopt another more formal approach.

In NuSMV, the system state is captured using *state variables* and the system state is changed by reassignment of the state variables with the *next* statement. NuSMV programs are structured into reusable *modules* for convenience purposes. For a complete explanation of the NuSMV input language, we refer the reader to [8] and the documentation for the NuSMV model checker. 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.

```
1  MODULE DeleteUser$0$(UserExists$0$)
2  VAR
3    return : boolean;
4  ASSIGN
5    init(return) := 0;
6    next(return) :=
7    case
8      (UserExists$0$ : 1;
9      1 : 0;
10   esac;
11   next(UserExists$0$) :=
12   case
13     (UserExists$0$) : 0;
14     1 : UserExists$0$;
15   esac;
16 FAIRNESS running;
17
18 MODULE VoidReceipt$0$0$(LoggedIn$0$, UserOfRole$0$2$,
19  ReceiptOpened$0$, ReceiptPosted$0$, ReceiptReversed$0$)
20 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 11-15, 30-35). In the example above, the passing of parameters into the use case modules is shown in lines 1, 18-19.

As can be seen in lines 3 and 21 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 5, 23) and if the pre-conditions of the use case are met then its value is re-assigned to 1 (lines 6-10, 24-29).

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. A process instance corresponding to a module is named with `activated_` prefixed to the name of the module. For example, the process instance for the module `DeleteUser$0$` is named `activated_DeleteUser$0$`. Using processes in NuSMV and including the FAIRNESS clause in the use case modules (lines 16, 36), 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 7. The mappings from structured use case models to NuSMV described in the previous section are used to translate the models created in the SUM Analyser 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 Analyser. 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.

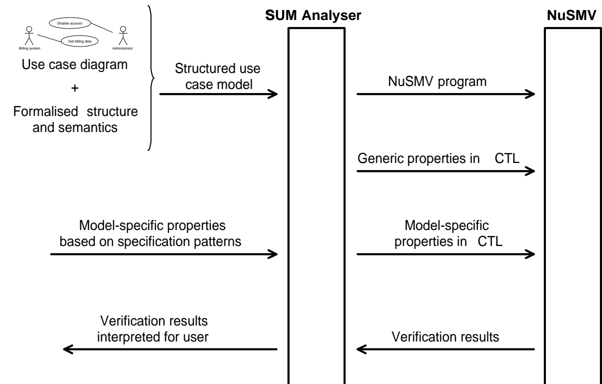

Figure 7. Verification with SUM Analyser

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" [17]. We define three liveness categories for a use case: *Dead*, *Transient* and *Live*. The SUM Analyser 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.

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)

Table 1. CTL Formulae for Generic Verification

Reversibility of conditions: The SUM Analyser 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.

Table 1 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$0$`.

Verification for liveness of use cases and reversibility of conditions with the SUM Analyser 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 Analyser. 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 Analyser pattern hierarchy is shown in Figure 8. The original patterns that were not included in our hierarchy are indicated with dashed lines and borders and the new patterns are shown with solid borders.

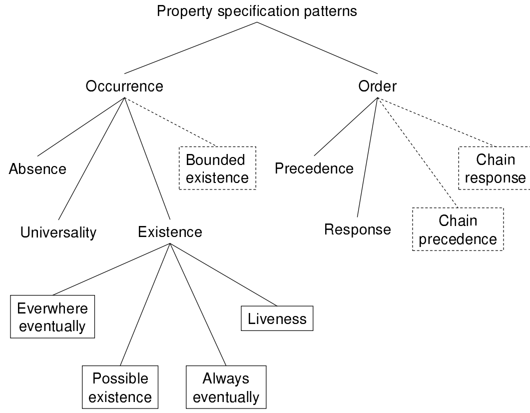

Figure 8. Property Specification Pattern Hierarchy for SUM Analyser

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 (→). For example, *User Exists (#jbloggs) & User Logged In (#jbloggs)* is a valid pattern variable for the CMS example. Each pattern is associated with a CTL formula, and once the user chooses a pattern and fills in the pattern variables, these are plugged into the CTL formula for that pattern. The resultant formula is a verification property that can be used for model checking.

Each of the specification patterns implemented in the SUM Analyser and shown in Figure 8 is described next. In the SUM Analyser, 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. These are shown in the Table 2. In the table, *v* stands for a pattern variable composed of condition instances and logical operators.

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" [17].

(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.

Existence pattern	CTL formula
Everywhere eventually	AF *v*
Possible existence	EF *v*
Always eventually	AG AF *v*
Liveness	AG EF *v*

Table 2. CTL Formulae for Existence Patterns

(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.

The NuSMV model checker generates a counter-example trace whenever the property is found to be false during verification. Such a counter-example trace shows a sequence of system state changes that lead to the property violation. During model-specific verification, such traces are interpreted for the user in terms of a sequence of use

case activations. However, certain properties such as *Possible existence* do not generate counter-examples. Properties created with the *Possible existence* pattern refer to a certain condition that needs to hold on at least one path of system execution. If such a property is found to be false, it means that such an execution path does not exist and essentially the collection of all the possible execution paths would comprise the counter-example. Determining the reason why such properties do not hold in a model becomes more difficult than with the availability of a counter-example trace. However, knowing that a certain property does not hold in a model can still be very valuable to the developer, as this identifies that there is an error in the model. In order to find the error, the developer should run verification on other related properties that can lead to the identification of the problem source.

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 and the following shows a condensed version of the counter-example trace was produced by the NuSMV model checker.

```
1  -- specification AG !(LoggedIn$0$ & !UserExists$0$)
2     is false
3  -- as demonstrated by the following execution sequence
4  -> State 1.1 <-
5    [executing process activated_AddUser$0$]
6      UserExists$0$ = 0
7      UserExists$1$ = 0
8      UserExists$2$ = 0
9      UserLoggedIn$0$ = 0
10     UserLoggedIn$1$ = 0
11     UserLoggedIn$2$ = 0
12     UserOfRole$0$0$ = 1
13     UserOfRole$0$1$ = 0
14     UserOfRole$1$0$ = 0
15     UserOfRole$1$1$ = 1
16     activated_LogIn$0$.return = 0
17     activated_LogIn$1$.return = 0
18     activated_LogIn$2$.return = 0
19     activated_AddUser$0$.return = 0
20     activated_AddUser$1$.return = 0
21     activated_AddUser$2$.return = 0
22 -> State 1.2 <-
23     [executing process activated_LogIn$0$]
24     UserExists$0$ = 1
25     activated_AddUser$0$.return = 1
26 -> State 1.3 <-
27     [executing process activated_DeleteUser$0$]
28     UserLoggedIn$0$ = 1
29     activated_LogIn$0$.return = 1
30 -> State 1.4 <-
31     [executing process activated_LogIn$1$]
32     UserExists$0$ = 0
33     activated_DeleteUser$0$.return = 1
```

In the above trace, lines 1 and 2 show the CTL formula that was violated in the model. The trace of the execution sequence violating the formula begins in line 4 and consists of several process executions. Each process execution (lines 5, 23, 27, 31) corresponds to a use case module instantiation, and hence can be interpreted as a use case instance activation. For example, `executing process activated_AddUser$0$` is interpreted as *Administrator.Add User (jbloggs)*. The fact that a use case instance is activated does not necessarily mean that the activation was successful. It is only successful in the case where the `return` variable for the corresponding module is assigned to `1` in the next state. In the trace above, we can see that in `State 1.2` the `return` variable for the `AddUser$0$` is indeed assigned to `1`, which is shown in line 25: `activated_AddUser$0$.return = 1`. Hence the activation of *Administrator.Add User (jbloggs)* was successful.

In the lines 6-21 the initial state of the system is given in terms of values of its state variables. After each process execution, only the state changes are given in the trace. For example, after the execution of `activated_AddUser$0$`, two state variables are changed: `UserExists$0$` and `activated_AddUser$0$.return` (lines 24, 25). In the final state of the system for this trace `LoggedIn$0$ = 1` and `UserExists$0$ = 0`, which violates the verification property.

The user of the SUM Analyser is sheltered from the details of interaction with the NuSMV tool. The NuSMV counter-example trace discussed above is interpreted in terms of use cases for the SUM Analyser user, in the following way.

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

The interpreted 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.

7. Evaluation of Case Study

We constructed structured use case models for the CMS requirements in the SUM Analyser. 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 and incomplete specifications were discovered and remedied during the process of merely structuring the use case descriptions according to the metamodel and syntax described in Section 4. For instance, many pre- and post-condition definitions were incomplete. Certain use cases had more than one flow, but only one set of associated pre- and post-conditions.

Numerous errors were identified in the CMS use case models by running them through the analyses in the SUM Analyser, examples of which were given in Section 6. 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.

The NuSMV model checker that we chose for this work performed relatively well in obtaining the analysis results for the CMS use case models. 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. For the CMS use case models, all verification results 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 [20].

8. Related Work

Several attempts have been made to address the drawbacks of use case modelling and formalise this method.

Hausmann *et al* [14] propose an approach to modelling and analysis of software requirements based on formalising relationships between several UML diagrams with graph transformation theory. This approach suggests that there are two main types of requirements models: static and dynamic. Static aspects of a system are modelled using class diagrams, while use case diagrams capture its dynamic requirements. Inconsistencies are often introduced when static and dynamic models are integrated, as currently there is no adequate mechanism to check an integrated model for consistency. Hausmann *et al* tackle this particular problem by defining explicit relationships between static and dynamic requirements models and proposing a means of analysing them for consistency.

Behaviour of individual use cases is described by activity diagrams in this approach. Activity diagrams give an overview of the sequential or branching flow of a set of operations within a system. For each operation appearing in an activity diagram, pre- and post-conditions are defined as collaborations. In UML, a collaboration refers to a set of classes or other elements that work together to achieve some common objective. Hausmann *et al* draw collaborations as object diagrams showing only the elements relevant to the particular operation. A pre-condition collaboration shows a snapshot of the system before the operation is executed, and a post-condition collaboration shows how the objects and their relationships change after the operation execution. These collaborations serve as a link between static and dynamic requirements models.

Graph transformation theory is used to formalise collaborations in models and this facilitates rigorous consistency analysis. The aim of the analysis is to uncover potential consistency problems and not to prove absolute model consistency. It is performed statically on the basis of *critical pair analysis*, and implemented in a tool called AGG. AGG is a tool for graph manipulations, and hence cannot be applied directly to UML models. For practical use of this approach, an interface between a UML tool and AGG needs to be defined and implemented. Such an interface would allow one to export UML models to AGG for analysis and then view analysis results in terms of the original UML models.

The method proposed by Hausmann *et al* is appealing, as it has the potential of allowing developers to build models using familiar visual techniques and at the same time benefit from formal analysis of these models. Similarly to our approach, use case and actor relationships are not directly supported by this method. However, we explain how the structured use case approach can be applied to existing use case models that contain such relationships by flattening.

Hausmann *et al* do not provide a means for handling relationships in existing use case models. Further, as presented in [14], the work is not substantiated with any application of the method in practice.

Back *et al* [4] formalise use case models with a precise mathematical notation called refinement calculus [5], which is an extension of Dijkstra's weakest precondition calculus. As in the method advocated by Hausmann *et al*, an effort is made to bring together modelling of classes and dynamic requirements for a system. Additionally, rigorous analysis for achievability of actor goals is proposed in this method.

According to this approach, classes with attributes and methods are defined in a formal textual notation. The collection of all class attributes describes the state of the system that can be changed by execution of use cases. Use cases are expressed as *contract* statements that essentially state how their execution affects the system state. Once again relationships between use cases are not taken into account by this method. In the prescribed notation, class and use case descriptions resemble computer programs.

Achievability analysis can be performed by first defining a goal formally and then performing weakest pre-condition computations to determine whether the goal can be achieved in the given model.

This approach certainly extends use case modelling with a formal notation and an analysis technique, however whether this is an enhancement to use case modelling is questionable: The work is currently not supported by any tool, which makes it difficult to judge the potential usability of the method. However, the nature of the underlying notation and analysis technique do not lend themselves to much automation, so our conclusion is that although interesting, this approach is impractical.

The structured use case modelling and analysis method that we propose is based on concepts similar to those used in the techniques developed by Hausmann *et al* and Back *et al*, such as pre- and post-conditions. However, it surpasses these techniques in improving use case modelling for two reasons. First, it maintains a relatively simple modelling notation while formalising use cases. Second, it is supported by a rigorous analysis technique and a tool that automates model analysis.

In their Masters dissertation [3], Andersson and Bergstrand formalise use case models with extended Message Sequence Charts (MSC) [15]. Their work focuses on developing a graphical yet formal notation for describing use case flows. There are no suggestions for analysing the proposed extended models. As for tool support, only several suggestions are given in the future work section of the dissertation.

In general, numerous efforts have been made to formalise different aspects of UML. The UML Version 2.0 [2] has been a work in progress by the Object Management Group for the past few years. The aim of UML 2.0 is to provide a more complete and formal specification of the language with a special emphasis on its semantics. With respect to use case modelling however, the new version of UML provides only a few insignificant changes.

The Object Constraint Language (OCL) [1] can be used to annotate certain UML diagrams with formal descriptions of constraints. OCL defines a relatively simple syntax and can be used to express invariants, queries, as well as pre- and post-conditions for UML modelling elements. The language is primarily based on the object-oriented concepts such as classes, associations and role names. Use case models deal with entities on a higher conceptual level, and hence applying OCL to use cases would not be practical. However, several constructs in our proposed notation for structured use case modelling resemble OCL.

With respect to the CMS case study, similar studies have been done using different approaches and tool support. In [13], Gimblett *et al* derive a formal specification of an electronic payment system in CSP-CASL from existing requirements documents. The original requirements specification consisted of textual requirements expressed in natural language, use cases and other UML diagrams. The authors report that the original specification was ambiguous due to the use of natural language, contained inconsistencies and was not suitable for automated analysis. The formalisation was performed manually and resulted in textual specifications of required system behaviour and relevant data types in CSP-CASL. The authors provide a reference to existing tools that can perform analyses such as consistency-checking and deadlock analysis. However, no concrete examples of what errors can be detected with automated analysis of the formalised electronic payment system are presented. Further, practical application of the approach presented in this case study is not evident, as it requires the developer to possess expert knowledge of the CSP-CASL formalism.

9. 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 Analyser 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 Analyser 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 OCL Final Adopted Specification, ptc/03-10-14. OMG Document, 2003.

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

[3] M. Andersson and J. Bergstrand. Formalizing Use Cases with Message Sequence Charts. Master's thesis, Department of Communication Systems at Lund Institute of Technology, Sweden, May 1995.

[4] 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.

[5] R.-J. Back and J. Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.

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

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

[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] A. Gimblett, M. Roggenbach, and H. Schlingloff. Towards a formal specification of electronic payment systems in csp-casl. In *Selected papers from WADT 2004. 17th International Workshop on Algebraic Development Techniques, Barcelona, Spain.*, LNCS 3423, pages 61–78. Springer, 2005.

[14] 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.

[15] ITU. Recommendation Z.120. Message Sequence Charts (MSC'96). ITU Telecommunication Standardisation Sector, Geneva, 1996.

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

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

[18] S. A. Kripke. Semantic Analysis of Modal Logic. In *I: Normal propositional calculi. Zeitsc. Math. Logik Grund. Math.*, pages 67–96, 1963.

[19] 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.

[20] 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.