Model Checking Visual Specification of Requirements

Ulka Shrotri, Purandar Bhaduri and R. Venkatesh
TRDDC, Tata Consultancy Services
54 B, Hadapsar Industrial Estate
Pune 411 013, INDIA
{ulkas,pbhaduri,rvenky}@pune.tcs.co.in

Abstract

Visual notations like class diagrams, and use case diagrams are very popular with practitioners for capturing requirements of software applications. These notations unfortunately have little or no semantics, and hence cannot be analysed by tools. Formal notations, on the other hand, have associated tools that check specifications for stated properties but are difficult to integrate with software development processes in use. Strengths of both approaches can be exploited by giving formal semantics to popular notations. Here we propose a novel usage of UML object diagrams for specifying pre- and post-conditions for use cases and capturing global system properties as class invariants. A translation is defined from object diagrams to the formal notation TLA$^+$. The TLA$^+$ specification is then formally verified using the model checker TLC. The proposed notation is intuitive, expressive and formal. We present a small case study to illustrate its strengths.

1. Introduction

In this paper, we apply model checking to the problem of verifying object oriented software requirements specifications. Formal specification languages like Z and VDM have been used to specify software requirements, but there are no fully automated tools for analysing these specifications as in model checking. On the other hand model checkers have been used to verify designs of control intensive software and hardware systems, and not for analysing requirements. We show that with the proper combination of specification language and model checking tool it is possible to check software requirements for consistency. Since requirements deal with data and states at a high level of abstraction, the method is more likely to scale up when confronted with industrial applications.

In object-oriented software development, the Unified Modelling Language (UML) [2] has become the industry standard for specifying different models. Use cases are the primary notation in UML for capturing requirements. Use case descriptions supplement the static domain model by describing the behaviour of the system using both dynamic and functional aspects. The dynamic aspect is the flow of actions of the system and the user, usually described by sequence or collaboration diagrams. The functional aspect is described by the pre- and post-conditions of the use case written in natural language. This functional aspect is not formally integrated with the static structure of the system described using class diagrams, which results in lack of tool support for detecting inconsistencies in use case descriptions.

In this paper, we propose a visual, yet formal, notation for capturing the relation between the functional and the static structure in requirements, and the use of a model checking tool to detect inconsistencies in their specification automatically. We assume that the entities in a use case are described by a class diagram. Corresponding to each class we associate an object diagram that specifies a global invariant satisfied by objects of that class. We also associate object diagrams with use cases for specifying their pre- and post-conditions. Our object diagrams are enriched with Boolean operators for combining simple conditions, and also have primitive attributes for specifying state changes in an object oriented system. The notation is formalised using the Temporal Logic of Actions.

The Temporal Logic of Actions (TLA) [9] and its associated specification language TLA$^+$ [10], provide a notation to specify a system as a set of actions. Each action effects a change in the global state. The same notation is also used to specify properties of the global state. This allows for a natural translation from use cases and global constraints in object-oriented models to actions in TLA$^+$. While translating from UML to TLA$^+$ we translate object diagrams associated with classes as invariant properties, and object diagrams associated with use cases into TLA$^+$ actions. Once models in UML are translated into TLA$^+$, we use the model checker TLC to catch errors in the translated models. We

demonstrate the translation to TLA$^+$ and the results of using TLC through a small case study, that of a library system. Our experience shows that the formalisation of object-oriented models in TLA and the use of TLC can catch many subtle errors in the initial specification [1].

Our choice of TLA and TLC for verifying object models over other formalisms was motivated by two key factors. Most model checkers allow only simple data types to describe states and focus exclusively on attacking the complexity arising out of composing several state machines in parallel. They do not provide the complex structural constraints on states that characterise software systems, object-oriented or otherwise. TLA, on the other hand, has a rich set of constructs for describing relationships between data, as well as transformations of these relations. On the other hand, formalisms like Z and VDM, which can describe relationships between data adequately, do not have fully automated analysis tools. Our formalisation of object-oriented requirements requires the combination of rich data constraints with the availability of automated tools, which is addressed by TLA and TLC to a nicety.

The main contribution of this paper lies in showing how visual notations familiar to practitioners can be used effectively to describe state properties of a system. An important advantage is that both static and dynamic aspects of models are specified using the same notation. We formalise the notation in TLA$^+$ allowing usage of tools like TLC to detect inconsistencies and violation of invariants. Through examples we demonstrate how it is intuitive and easier to specify constraints in our notation when compared with the Object Constraint Language (OCL) [14]. OCL is a textual language for expressing constraints on UML models. OCL has two major drawbacks which we address using our formalised object model notation. The first is the limited availability of tools for the automatic verification of constraints against model diagrams. The second is the difficulty of using a textual formalism in conjunction with the visual languages used in the rest of UML. This difficulty is acutely evident when trying to parse OCL navigation expressions, one reason why OCL is not widely used among practitioners. These comments are based on the current official OMG version of UML 1.4 [11]. It remains to be seen how the draft UML 2.0 addresses some of these issues.

Related Work The specification of actions by snapshots of object configurations before and after an operation is used by Catalysis [5]. But this notation is used as an informal documentation tool without any formal semantics. Moreover, snapshots in Catalysis are concrete instance states, rather than properties of states. The work on constraint diagrams [8] and spider diagrams [7] allows pre/post-conditions and invariants to be expressed visually, using variants of Euler and Venn-Pierce diagrams and involve the use of 3D notation to specify constraints. Since these diagrams are not based on the graphical elements of UML, they place the burden of a new and unfamiliar notation on the average software developer. The visualisation of OCL constraints using extended collaboration diagrams explored in [3, 4] also uses collection of objects and links between them to express properties or constraints. However, their approach focuses on the use of graph rewriting with rule expressions to provide a semantics for OCL. The story diagrams of FUJABA [6], used for specifying pre- and post-conditions of methods, are similar to our object diagrams. The FUJABA tool translates specifications using story diagrams to Java code. In contrast to all of the above works, our proposal is to detect inconsistencies in visual requirements specifications automatically using an available model checker. We do this by translating specifications in a UML-based visual constraint definition language into a temporal-logic formalism.

2. The Temporal Logic of Actions

The Temporal Logic of Actions (TLA) was proposed by Lamport [9] as a logic for specifying and reasoning about concurrent systems. TLA uses a single logical formalism for describing transition systems and formulating their properties. It is an extension of Linear Time Temporal Logic to a *logic of actions*. TLA$^+$ [10] is a complete specification language based on the logic of TLA.

In TLA actions specify changes in state. A *state* is an assignment of values to program variables. The value assigned to variable x by state s is written $s[\![x]\!]$. A *state function* f is an expression over program variables, which evaluates to $s[\![f]\!]$ in state s. *State predicates* are Boolean valued state functions.

In TLA, a behaviour σ is an infinite sequence $\langle s_0, s_1, s_2, \ldots \rangle$ of states. A terminating behaviour is a special case which repeats the final state forever. A pair of consecutive states (s_i, s_{i+1}) in σ is called a *step*. A state predicate $P(u)$, involving program variables u, is true for a behaviour σ if it holds in the initial state s_0 of σ, i.e., $s_0[\![P(u)]\!] = \text{TRUE}$.

The formulas of TLA are built from actions, Boolean operators, and the temporal operator $\Box$. The temporal formula $\Box P(u)$ is defined to be true if $P(u)$ holds for all suffixes of σ, including σ itself. An *action* A is a Boolean expression containing primed and unprimed variables. For states s and t, $[\![A]\!](s,t)$ is defined to be TRUE iff A holds with values from s substituted for unprimed variables and with values from t substituted for primed variables. The action A is considered a temporal formula by letting $[\![A]\!](s_0, s_1, s_2, \ldots)$ equal $[\![A]\!](s_0, s_1)$. For any state function f, let $[A]_f \triangleq A \vee (f' = f)$, where f' is the expression obtained by priming the free variables in f. A step satisfies $[A]_f$ iff it sat-

isfies A or it leaves f unchanged (a "stuttering" action). The formula $\Box[A]_f$ asserts that every step is an A step or leaves f unchanged. The canonical form of a TLA formula is $Init \wedge \Box[N]_f \wedge F$, where $Init$ is a state predicate describing the initial states of the system, N an action, f a state function, and F is a *fairness condition*. In this paper we do not make use of fairness specifications and all our specifications have the form $Init \wedge \Box[N]_f$.

3. Modelling Requirements in UML

A software system can be modelled as a set of interactions with the environment. Each interaction changes the state of the system. The requirements model of such a system will include a model of interactions between the system and the environment, a model of the state and a set of rules that the system must obey. In UML these can be modelled using use-cases, class diagrams and constraint specifications respectively. There are other aspects to the requirements of a software system like business processes; these are outside the scope of this paper.

In UML constraints can be specified in the following ways.

Global Invariants These are rules that hold within the system at all times. Association constraints – cardinality and optional/mandatory specifications – are one way of specifying them. Global invariants can also be stated by using a constraint language like OCL. We show how object diagrams can be used for this purpose.

Rules of Interactions The interactions between the system and the environment can be constrained by associating pre- and post-conditions with use cases. These conditions can be specified in UML, using OCL. We specify both pre- and post-conditions using a single object diagram in a clear and concise way.

Class Invariants In UML these can be specified by associating an OCL expression with a class. Here we illustrate how object diagrams can be used for this purpose.

4. Semantics for UML Class and Object Diagrams

In this section we describe a translation from UML object and class diagrams to TLA$^+$ thus giving these diagrams an implicit semantics.

Classes and Objects A class type is specified in TLA$^+$ as a set of records with the field names specifying the attributes of the class. Every class gives rise to a set of objects of that class. Every element of this set, i.e., an object of this class, ranges over the set of records corresponding to the class. Since objects can be created or destroyed, a class is modelled as a variable set. For convenience, we use C_i to refer to a UML class as well as the corresponding set of records in TLA.

Types and Data Invariants Since TLA does not have the notion of typing, the type of each attribute and the data invariants can be captured by straightforward invariants in the TLA$^+$ specification.

Associations and Links In TLA$^+$ links between objects are specified as variables that range over relations (sets of tuples) between appropriate sets. The cardinality and optionality constraints on associations are also captured as invariants.

Object Diagram An object diagram is a graph of instances. The vertices are objects with data values and the edges are links, which are instances of associations in the class diagram. A static object diagram is an instance of a class diagram; it is intended to show a snapshot of the detailed state of a system at a point in time. Here we give a more general semantics to object diagrams extended with logical operators, and use them to specify pre-, post-conditions and invariants. While modelling requirements, an extended object diagram may be attached to a class or a use case. When attached to a class it specifies a class invariant. When attached to an use case it specifies the pre- and post-condition for the use case. In both cases, an extended object diagram can be used to represent constraints on the existence of given objects and links as well as data attributes of instances of a class.

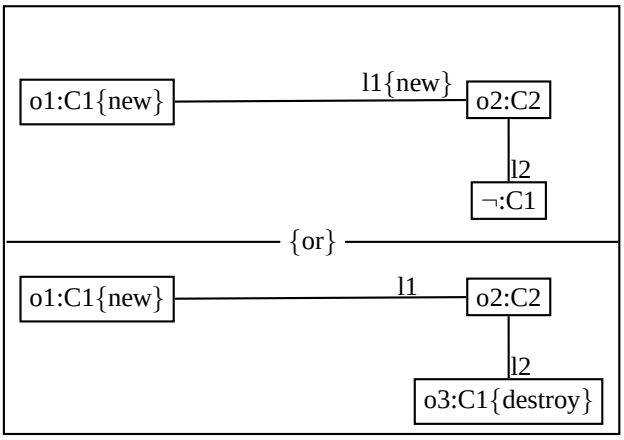

Figure 1. Example Object Diagram

Figure 1 shows an example object diagram that is attached to a use case with o2 as a parameter. As it is a composite diagram with an {or} connector, the pre-condition

of the use case is a disjunction of two conditions. The first condition says that there must be no instance of *C1* linked to `o2` by `l2`. The second disjunct states that such an object `o3` indeed exists. The post-condition in each case is indicated by the attributes {`new`} and {`destroy`}. If the first pre-condition holds, then an object `o1` will be created and linked to `o2` by the link `l1`. If the second pre-condition holds then the object `o3` and link `l2` are destroyed, a new object `o1` is created, and a new link `l1` is set up between `o1` and `o2`. Note that we have not tagged the link `l1` with {`new`} in the second diagram, as this can be inferred from the annotation {`new`} on `o1`. The tags {`new`} and {`destroy`} on a link are optional if the link connects an object that is created or destroyed.

The notation is quite expressive. It can be used to express universal properties like "all books of a given title `t` have been loaned". This is shown in the object diagram in figure 2, which says "there is no book with title `t` that is not loaned."

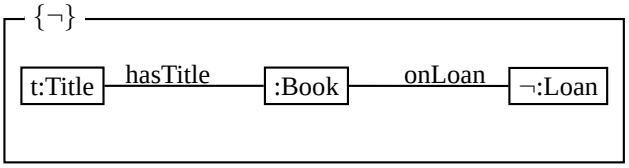

Figure 2. A Universal Property

Informally, the algorithm to translate an object diagram to TLA$^+$ is as follows.

An object diagram is either simple or composite. A composite object diagram is either a ¬ of a diagram or two diagrams combined by {`or`} or {`and`}. Each of the component diagrams is translated to a TLA$^+$ predicate and these are combined using the corresponding boolean operator in TLA$^+$.

We now describe how a simple object diagram, such as the top or bottom parts of Figure 1, is translated to TLA$^+$. A simple object diagram is a graph $< O, L >$ with objects O as vertices, and links L as edges. The vertices and edges may be additionally decorated with an attribute from the set {`new,destroy,opt,`¬}. The attribute {`opt`} is just a shorthand for combining simple diagrams involving optional cases connected by an {`or`}. Therefore, we treat {`opt`} as syntactic sugar and leave it out of this discussion. We view a simple object diagram as stating a boolean condition on the state of the system. The attribute {`new`} is not relevant in evaluating this condition, as it refers to the post-state of a use case. The condition asserted by a simple diagram is translated into TLA$^+$ as the assertion that a certain set of tuples of objects defined by the diagram is non-empty. More formally, given an object diagram $< O, L >$, let $O = O_1 \cup O_2$ and L = $L_1 \cup L_2$ be partitions of O and L, where O_1 and L_1 consist of objects and links that are either undecorated or decorated with {`destroy`} and O_2 and L_2 consist of objects decorated with {`new`}. In TLA$^+$ we define a set of tuples $< o_1, \ldots, o_k >$ consisting all objects in O_1, constrained by the following conditions:

1. Each o_i to belong to set C_i where o_i is an instance of UML class C_i.
2. For each link l_i in L_1, that connects two objects o_l and o_m, in O_1 we require that the tuple $< o_l, o_m >$ belong to the set l_i.
3. The negation ¬`:C` requires the non-existence of any object of class `C` having the associated links.

If this set of tuples $< o_1, \ldots, o_k >$ is non-empty, we say that the condition stated by the diagram is true of the current state.

In addition to describing a condition on a state, an object diagram also describes the post-state of a use case, provided the pre-condition of the use case holds. The post-state is described using the attributes {`new`} and {`destroy`}. Objects tagged with {`new`} result in the addition of a new object in the post-state to the set corresponding to its class. All links with that object as an end point update the corresponding sets. All objects and links tagged with {`destroy`} are removed from the corresponding sets in the post-state.

The general translation scheme is illustrated through the library case study described below. The case study also demonstrates how object diagram as a notation and its translation to TLA$^+$ can be used for the specification and analysis of business applications.

5. Case Study: A Library System

We now formally describe using TLA$^+$, a library system specified using UML and the Catalysis notation in [13]. This example highlights the object diagram notation and its translation to TLA$^+$, as well as the advantages of early verification using model checking. Indeed many subtle errors in our initial specification of the library system were detected by the use of the model checker TLC.

5.1. Informal Description

We start with an informal description of the library system as presented in [13]. This description had gaps and ambiguities, that were uncovered by our formal visual specification and the use of the TLC model checker.

A library maintains a collection of books. Members belonging to the library borrow and return books. A member may also reserve a title if all books bearing that title have been issued to other members. On return of a book, if there

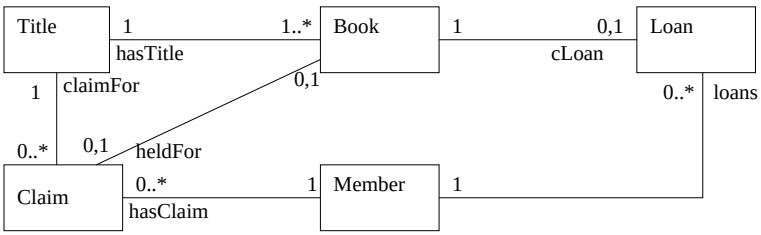

Figure 3. Class Diagram for Library

MODULE Relations

Defines binary relations between sets with cardinality and optionality constraints, for defining associations between classes.

$Rel(X,Y) \triangleq SUBSET\ (X \times Y)$ *SUBSET A is the set of all subsets of A.*

$First(R) \triangleq \{t[1]: t \in R\}$
$Second(R) \triangleq \{t[2]: t \in R\}$

$$OneToOne(R) \triangleq \land\ \forall\ x \in First(R),\ y_1,y_2 \in Second(R) : \langle x,y_1\rangle \in R \land \langle x,y_2\rangle \in R \Rightarrow y_1 = y_2$$
$$\land\ \forall\ y \in Second(R),\ x_1,x_2 \in First(R) : \langle x_1,y\rangle \in R \land \langle x_2,y\rangle \in R \Rightarrow x_1 = x_2$$

$$OneToMany(R) \triangleq \forall\ y \in Second(R),\ x_1,x_2 \in First(R) : \langle x_1,y\rangle \in R \land \langle x_2,y\rangle \in R \Rightarrow x_1 = x_2$$

$$ManyToOne(R) \triangleq \forall\ x \in First(R),\ y_1,y_2 \in Second(R) : \langle x,y_1\rangle \in R \land \langle x,y_2\rangle \in R \Rightarrow y_1 = y_2$$

$MandatoryFirst(R,X,Y) \triangleq \forall\ y \in Y : \exists\ x \in X: \langle x,y\rangle \in R$
$MandatorySecond(R,X,Y) \triangleq \forall\ x \in X : \exists\ y \in Y: \langle x,y\rangle \in R$
$MandatoryBoth(R,X,Y) \triangleq MandatoryFirst(R,X,Y) \land MandatorySecond(R,X,Y)$

Figure 4. The Relations Module

are pending claims for that title, then the book is held for one of the members having a claim. Members may also cancel their claims.

A class diagram for the library system is shown in Figure 3. The meaning of the associations is as follows:

`hasTitle(bk,ttl)` The title of book `bk` is `ttl`.

`cLoan(bk,ln)` Book `bk` is currently loaned on loan `ln`.

`loans(mem,ln)` Loan `ln` belongs to the set of loans of member `mem`.

`hasClaim(mem,clm)` Claim `clm` belongs to the set of claims of member `mem`.

`heldFor(bk,clm)` Book `bk` is held for claim `clm`.

`claimFor(clm,ttl)` Claim `clm` is for title `ttl`.

The classes and associations of the library system shown in Figure 3 are modelled in TLA$^+$ using variables as follows.

MODULE Library

TLA specification for the library system

```
EXTENDS Relations, Naturals, FiniteSets
```

Variables for names of classes

```
VARIABLES Book, Title, Member, Claim, Loan
```

Variables for Associations

```
VARIABLES heldFor, hasTitle, claimFor, hasClaim
VARIABLES loans, cLoan
```

The cardinality and optionality requirements on the associations in the class diagram in Figure 3 are captured by the invariants shown in Figure 5. The required definitions are in the module `Relations` in Figure 4.

The invariants of the library system consist of the invariants on the cardinality and optionality of the associations in Figure 3 together with an invariant on class `Book`: if a book `bk` is not loaned to any member and there are claims against the title `t` of the book for which no book is held, then the book must be held for some claim `c`. This is shown visually by the object diagram in Figure 6, which is attached to the class `Book`. Informally the diagram asserts that a particular set of 3-tuples is empty (indicated by the $\neg$ at top left

Association Invariants

Type invariants for associations expressing cardinality and optionality. Rel, OneToOne, ManyToOne, OneToMany, MandatoryFirst and MandatorySecond are declared in module Relations.

```
AssociationInvariants ≜
 ∧ heldFor ∈ Rel(Book,Claim) ∧ OneToOne(heldFor)
 ∧ hasTitle ∈ Rel(Book,Title) ∧ ManyToOne(hasTitle) ∧ MandatoryBoth(hasTitle,Book,Title)
 ∧ claimFor ∈ Rel(Claim,Title) ∧ ManyToOne(claimFor)
    ∧ MandatorySecond(claimFor,Claim,Title)
 ∧ hasClaim ∈ Rel(Member,Claim) ∧ OneToMany(hasClaim)
    ∧ MandatoryFirst(hasClaim,Member,Claim)
 ∧ loans ∈ Rel(Member,Loan) ∧ OneToMany(loans) ∧ MandatoryFirst(loans,Member,Loan)
 ∧ cLoan ∈ Rel(Book,Loan) ∧ OneToOne(cLoan)
    ∧ MandatoryFirst(cLoan,Book,Loan)
```

Figure 5. Invariants on Associations

of the figure), with each tuple consisting of a book `bk`, a title `t` and a claim `c1`. These tuples satisfy the constraints that there exists a link `hasTitle` between `bk` and `t`, a link `claimFor` between `c1` and `t` and no link `heldFor` between a book object and claim `c1`. The diagram also says that there does not exist a claim object (indicated by the ¬ prefixing `:Claim`) having a link `heldFor` with `bk`, nor does there exist any loan object linked to `bk` by `cLoan`. The TLA$^+$ specification corresponding to this rule is given below.

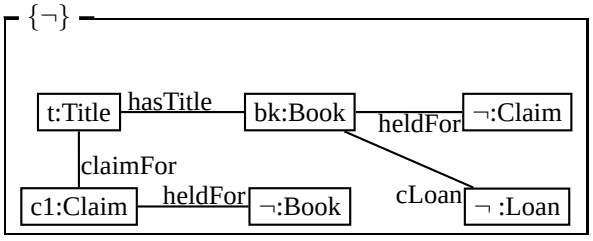

Figure 6. Invariant on Class Book

Claim Rule

Invariant on class book

```
BookInvariant ≜
∀ b ∈ Book :
LET
tuple1 ≜
{ <bk,t,c1> ∈ Book× Title × Claim :
   ∧ bk = b
   ∧ <bk,t> ∈ hasTitle
   ∧ <c1,t> ∈ claimFor
   ∧ ¬∃ c ∈ Claim : <bk, c> ∈ heldFor
   ∧ ¬∃ l ∈ Loan : <bk,l> ∈ cLoan
   ∧ ¬∃ bk1 ∈ Book : ∧ <bk1,t> ∈ hasTitle
                     ∧ <bk1,c1> ∈ heldFor
}
IN Cardinality(tuple1) = 0
```

Our library system consists of four use cases - Borrow, Return, Reserve and Cancel. For each use case we give an informal description of the functionality followed by the visual specification. Due to lack of space we omit the corresponding TLA$^+$ specification, except for the use case Borrow.

Borrow(m,b) Any member `m` can borrow a book `b` as long as it is not loaned to or held for any other member. Once a book has been loaned to a member the `cLoan` association is updated appropriately. The visual specification of this is shown in Figure 7. The specification states that a new loan object will be created along with the associations shown and the claim object `c1` will be destroyed at the end of this use case. The corresponding TLA$^+$ specifications is given below.

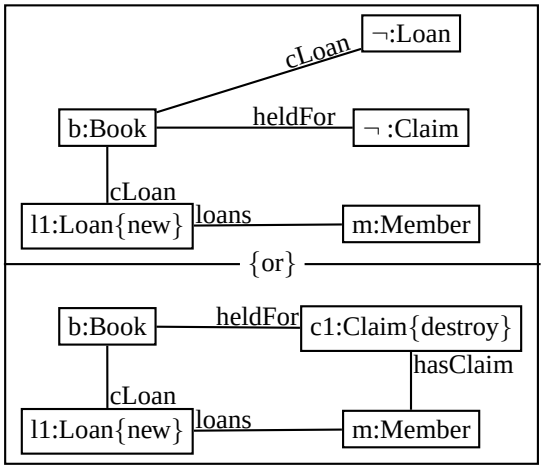

Figure 7. Object Diagram for Borrow

Borrow Use Case

TLA specs for borrow use case

```
Borrow1(m,b) ≜
LET
tuple ≜
{ <bk,m1> ∈ Book× Member :
   ∧ bk = b
   ∧ m1 = m
   ∧ ¬∃ c ∈ Claim : <bk, c> ∈ heldFor
   ∧ ¬∃ l ∈ Loan : <bk,l> ∈ cLoan
}
l1 ≜ CHOOSE l ∈ allLoans : l ∉ Loan
IN
∧ Cardinality(tuple) > 0
∧ Loan′ = Loan ∪ {l1}
∧ cLoan′ = cLoan ∪ { <b,l1> }
∧ loans′ = loans ∪ { <m,l1> }

Borrow2(m,b) ≜
LET
tuple ≜
{ <bk,m1, c1> ∈ Book× Member × Claim:
   ∧ bk = b
   ∧ m1 = m
   ∧  <bk,c1> ∈ heldFor
   ∧  <m1,c1> ∈ hasClaim
}
l1 ≜ CHOOSE l ∈ allLoans : l ∉ Loan
IN
∧ Cardinality(tuple) > 0
∧ Loan′ = Loan ∪ {l1}
∧ cLoan′ = cLoan ∪ { <b,l1> }
∧ loans′ = loans ∪ { <m,l1> }
∧ Claim′ = Claim \
{c1 ∈ Claim: <b,m,c1> ∈ tuple}
∧ heldFor′ = heldFor \
  { <b,c1> ∈ heldFor : <b,m,c1> ∈ tuple}
∧ claimFor′ = claimFor \
  { <c,t1> ∈ claimFor : <b,m,c> ∈ tuple }
∧ hasClaim′ = hasClaim \
  { <m,c> ∈ hasClaim :  <b,m,c> ∈ tuple}

Borrow(m,b) ≜
Borrow1(m,b) ∨ Borrow2(m,b)
```

Return(m,b) A member `m` may return a book `b` that is issued to her. On return the book becomes available to other members, unless there is an unfulfilled claim against the title of the book. An unfulfilled claim is one with no book being held for it. If there is such a claim then the book is held for it. The corresponding object diagram is shown in Figure 8. The {`new`} tag added to heldFor states that this link will be created at the end of this use case. The {`opt`} tagged to object `c` indicates that this object may or may not exist. This is easier than combining specifications using {`or`}.

Reserve(m,t) If no book of a particular title `t` is available for borrowing a member `m` may opt to make a claim against that title. The visual specification is shown in Figure 9.

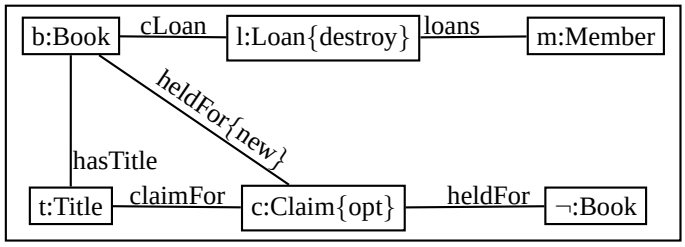

Figure 8. Object Diagram for Return

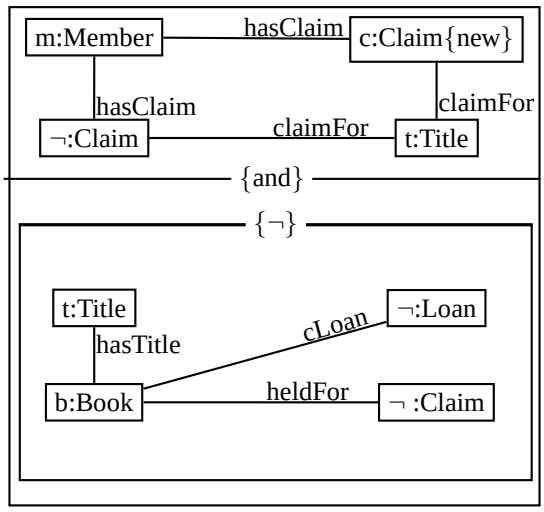

Figure 9. Object Diagram for Reserve

Cancel(m,t) A member `m` who has a claim against a title `t` can cancel his claim. If at the time of cancellation a book is held for him, then the book should be held for some unfulfilled claim against the title, if any; if there is no such claim, then the book held for this member should become available. The object diagram for the specification is shown in Figure 10.

Analysis with TLC After translating the specification to TLA$^+$ we ran the model checker TLC on the resulting specification. Several errors were detected in the initial specification of the library system by running TLC. All these errors were caught as instances of invariant violation. By inspecting the error trace generated by TLC we were able to locate the source of the error. We list two of these errors by stating them as requirements which were missing in our original specification, not presented in this paper.

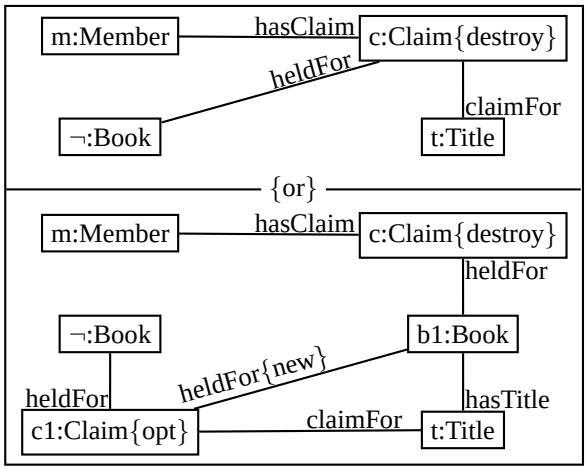

Figure 10. Object Diagram for Cancel

1. In the invariant shown in Figure 6, the initial specification did not require that there be no other book held for the claim against the title `t`, i.e., the lower-left `heldFor` link was missing.

2. In the step `Cancel(m,t)`, if a book with title `t` is held for member `m`, and there are more unfulfilled claims for the title, the book should be held for one of them.

In order to catch these errors, it was sufficient to run TLC on a model with just four books with three titles and four members. The set of all loans and claims were also limited to sizes of four and ten. It took TLC one minute and forty seconds on a 1.4 GHz Pentium 4 machine, with 512 MB RAM, to detect that the model presented here is error free. The presence of errors is usually detected in a much shorter time. We believe that these results are encouraging, even though the object model itself is quite small.

6. Conclusion

We have presented a visual notation with associated formal semantics and shown its usefulness in the rigorous analysis of requirements specified using UML. The main contribution of this paper is the combination of UML visual syntax with model checking based automated analysis to check for consistency in requirements. We have tried the approach on a real life financial negotiation system with 14 classes and 38 invariants. The number of constraints involving associations were much higher than the those involving class attributes, which were also modelled. We found errors in the specification by model checking the specification of one complex use case. Based on this experience we believe that the method will scale. As future work, we plan to integrate this method with the UML modelling tool MasterCraft [12].

7 Acknowledgements

We thank Prof. Mathai Joseph for his guidance and encouragement throughout the project. Thanks are also due to Ashok Sreenivas, Prahlad Sampath and Sreedhar Reddy for giving us valuable comments on the draft.

References

[1] P. Bhaduri, R. Venkatesh, and P. Sampath. Specification and verification of object models with TLA and TLC. Technical report, TRDDC, Mar. 2003.

[2] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[3] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. Consistency checking and visualization of OCL constraints. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000*, volume 1939 of *LNCS*, pages 294–308. Springer, 2000.

[4] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. A visualization of OCL using collaborations. In M. Gogolla and C. Kobryn, editors, *UML 2001*, volume 2185 of *LNCS*, pages 257–271. Springer, 2001.

[5] D. D'Souza and A. Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley, 1998.

[6] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph transformation language based on UML and Java. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. Theory and Application to Graph Transformations*, volume 1764 of *LNCS*. Springer, 1998.

[7] J. Howse, J. Molina, F. Taylor, S. Kent, and S. Gill. Spider Diagrams: A Diagrammatic Reasoning System. *Journal of Visual Languages and Computing*, pages 299–324, 2001.

[8] S. Kent. Constraint Diagrams: Visualizing Assertions in Object-Oriented Models. In *OOPSLA'97*, pages 327–341. ACM Press, 1997.

[9] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[10] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[11] OMG. The Unified Modeling Language (UML) Specification - Version 1.4, Sept. 2001. Joint submission to the Object Management Group (OMG) `http://www.omg.org/technology/uml/index.htm`.

[12] Mastercraft. Tata Consultancy Services. `http://www.tata-mastercraft.com`.

[13] Business modelling study – the library. TriReme International Ltd. white paper. Available at `http://www.trireme.com/whitepapers/process/business_modelling.html`.

[14] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.