Object-Oriented Design, Formal Verification, and Implementation of a Microkernel

Ali Ghanbari and Alireza Bagheri
Amirkabir University of Technology
ali.ghanbari@aut.ac.ir, ar_bagheri@aut.ac.ir

Abstract

Undoubtedly the development of formally verified software systems is time consuming. We can refer to the seL4 microkernel, a fully formally verified OS kernel, as an example that its specification and verification took a long time. The time and effort can be more than this, depending on the development team size, and also on the proficiency of team members in using formal methods.
In this paper, we propose a method to increase the productivity of formally verified microkernels development. Our approach is to use the UML in conjunction with B-Method. We present our solution by inspecting the development process of a simple microkernel, named *Platypus*. In the project Platypus, we have proposed a systematic, as well as feasible approach to translate UML models into B models, thereby letting us formally verify our UML models.

Keywords
UML, OCL, B-Method, AtelierB, Microkernel, Algebra

1 Introduction

1.1 Rationale

The seL4 microkernel is the first OS kernel that has ever been designed and verified completely formal [1]. It took a long time to develop seL4 [1]. The time can even be more than this, depending on the number of engineers in a team and the team members proficiency in using formal methods. Furthermore, the smallest change in the specification of the system, depending on its effect on the system invariant maybe costly to do. The long time to develop formally verified systems, and their costly change embodied our motivation to start the project Platypus.

In this project, we have proposed a method to increase the productivity of formally verified microkernels development. Our approach of increasing the productivity of microkernels development, is to use the UML [2] in conjunction with the B-Method [3]. A great deal of effort has been done over years to use the UML in conjunction with formal modelling languages [4, 5, 6, 7]. Unfortunately, applying many of these methods is not practical. This is because the UML, except for the OCL [8], basically lacks enough precision to be used with formal modelling languages, and any attempt in order to increase the precision of the UML, due to extensive use of constraints, would make understanding, analysis, and development of models too complicated. Hence, these attempts sometimes are considered almost futile [9]. Our approach, compared to similar approaches, is a systematic, as well as feasible approach to translate UML models into B models.
Depending upon our framework, we have developed a simple microkernel from scratch, approximately in four months. We have formally verified its correctness. Development and formal verification of a microkernel by a small team, indicates that our approach to use the UML in conjunction with B is practical. In spite of poor performance efficiency of the Platypus, it is possible to achieve acceptable efficiency by virtue of good kernel design [10].
The advantages of object-oriented design and analysis are evident for software engineers. Object-oriented design techniques tend to be used widely in modern operating systems [11]. By using object-oriented techniques we are able to reduce problem complexity, and also to minimise change costs.
We used the B-Method in this project, because there exist two industrial software tools that support the B-Method. By the use of tools supporting the B-Method, we are able to formally verify and refine systems with higher speed and more accurately. It has been experienced that in the course of system specification change, we can use many the proofs

of already discharged proof obligations to discharge new ones. Furthermore, the B-Method covers software development process from abstract specification, to an implementation through successive refinement steps [3], so we can better take care of performance efficiency considerations in the refinement steps. These are the main advantages of B over other formal methods, such as using Z or VDM.

1.2 Related Works

From a software engineering point of view, there are many projects proposing methods similar to one proposed in Platypus [4, 5, 6, 7]. With a critical appraisal on these works, we notice that all of the existing methods compel a software engineer to make UML models full of constraints, which in practice, especially in the early phases of system modelling, is very difficult to take care of all these details. In fact, all of the solutions that are used to increase the precision of the UML, or attempts that are made to translate UML models to formal models, are in opposition with the basis of the UML—namely, facilitation of specification through being informal. This is because some of the people in software engineering research community consider these works fruitless in practice [9]. In project Platypus, we tried to translate both behavioural and structural models, into B without any change and extension of the B-Method and with a minimum use of constraints.
Based on our knowledge, from an operating systems design and implementation point of view, project Platypus is the first ever microkernel that has been designed through object-oriented design techniques, using UML, and formally verified. Nevertheless, as we know, we should take special care about kernel design in order to be able to claim that the resulting microkernel will perform efficiently on the host machine; for example, address space switching and IPC implementations should be tailored for a specific host machine [10]. Hence, in order to use the Platypus microkernel, we should do a lot to revise its design and optimise its performance; in the case of the seL4 microkernel, in order to upgrade its performance to an acceptable amount, the seL4 team thought out ways to optimise kernel performance on ARM microprocessors [1].

1.3 The *Atelier B* Tool

The Atelier B [12], as computer software, developed by ClearSy company, can be used to support the B-Method. This tool is developed to be used in the industry [4]. The Atelier B has numerous tools such as a powerful editor with the ability to warn the user in the case of mistyping or even potential typing errors, automatic proof obligation generator, automatic prover, and an interactive prover. The automatic prover of the Atelier B is very effective; we experienced that the automatic prover discharges, in average, more than half of the proof obligations. Of those remaining unproved proof obligations, only a small number of them need more time to get proved, and usually they get discharged easily. In the case of the Platypus development team, the proof speed was approximately 20 proof obligations per day; this amount would definitely be increased with more people contributing in the team, or having more experience of using B-Method.
The remainder of this paper, is organised as follows. In Section 2, we describe our methods for translating UML models into B models. Section 3, is about the development process of the Platypus. In Section 4, we present a whole view of the Platypus. And finally Section 5, contains the conclusion of our efforts.

2 Transforming UML Models into B Models

Generally, we can classify diagrams of UML models into two main types—*behavioural* and *structural* diagrams [2, 13]. By using behavioural modelling constructs of the UML, we can model dynamic aspects of our systems, by describing a sequence of instructions, or a sequence of interactions between objects. At the same time, by the use of structural modelling constructs of the language, we can model static aspects of a system using classes and their relationships with each other, and also using specification of different components of the system. In the project Platypus, we propose methods for transforming both behavioural and structural UML models into B formal models. We choose Class Diagram from structural modelling, and Activity Diagram from behavioural modelling constructs of the UML to be transformed. We do not propose any method for transforming those parts of the UML, that are not used in our project.

2.1 Behavioural Modelling

In fact, the hardest part of the process of translating a UML model into a formal model is the translation of behavioural diagrams. Of those diagrams for behavioural modelling, we choose activity diagrams for being transformed into B; in the project Platypus we have used activity diagrams for modelling the

behaviour of system calls. Typically, UML activity diagrams contain components such as initial node, final node, control flow paths, actions, decision nodes, and merging nodes. Constructs such as fork and join are also used to model parallel behaviours, but since we have not used any of these constructs, we have not proposed any method for translating them.

Informally, we can describe the semantics of activity diagrams in the following way. In the first place, the initial node produces a *token*. Once this token reaches an action, the action consumes the token and produces another token instead. This newly created token goes down through the output control flow of the action. The consumption of a token by an action is equivalent to the execution of that action. The execution of an action means the execution of the instruction written on that action in a given activity diagram. These instructions, by following which a transition in the system state occurs, are usually described using natural language. But the description using natural language would cause several problems in the transformation of UML models into formal models, because natural language terms may be ambiguously stated, or cannot be precise enough [14]. In order to avoid ambiguous description, and to have a precise description, we have to use a formal specification language for describing the desired state of the system after the execution of an action. In this project we have used OCL as a formal specification language.

The main idea behind the precise development of activity diagrams is to decompose substantial actions into fine grained actions and to describe them using more primitive actions. If we describe the side-effects of performing each action using an OCL expression, we can describe the side-effects of carrying out actions in an activity by accumulating these side effects and representing them using a conjunction of predicates. When we start to develop an activity diagram, we usually cannot directly describe the side-effects of a substantial action using OCL predicates in the first place, so we should decompose these actions and describe them using sub-activity diagrams. If we continue to decompose actions gradually, we can ultimately describe complex actions using primitive actions, describing the side effects of which are easy.

In this section, we explain our idea by precisely developing the activity diagram of the system process scheduler, and introducing several guidelines to translate these kinds of models into B. The capability of scheduling processes is a functional requirement of the Platypus. We are going to describe the behaviour of the system as the process scheduling use-case is activated. At the first step, the only thing we know about of the use-case, is its activation precondition. The system scheduler is activated by an external interrupt, so its activation should not depend upon the satisfaction of certain condition. The side-effects of this use-case remains unspecified, because we are not sure about its details. Thus, we have to decompose the Do Schedule action in the activity diagram of Figure 1, into more simple actions. We use a *rake* symbol, next to an action, to represent that there is a sub-activity diagram refining the action.

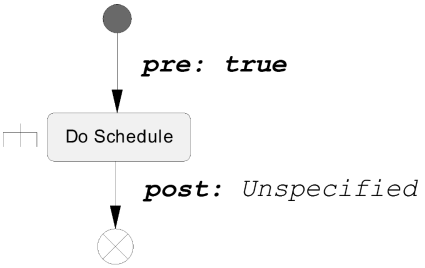

Figure 1: Preliminary activity diagram for scheduling processes, which its post-condition left unspecified.

In order to specify the action Do Schedule more precisely, we use a sub-activity diagram. The sub-activity diagram is shown in Figure 2[1]. As we can see, we have specified the action in some more details.

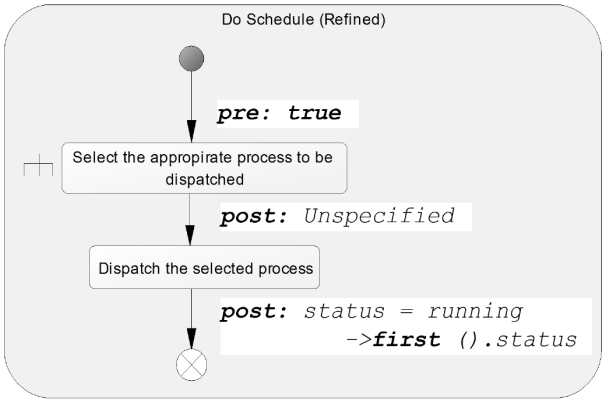

Figure 2: Detailed sub-activity diagram, refining the action Do Schedule.

As before, this diagram contains an action which we are not able to specify precisely. So we need to specify the unspecified action using another sub-activity

[1]Note that, by **status**, we mean the essential information to make the host machine run a process.

diagram. If we continue to decompose complex actions and refine them, we reach an activity diagram that is shown partially in Figure 4. The actions in this diagram are simple enough, and we can specify them directly using OCL.

During the specification of the system scheduler, we assumed that the state of a process object can be transitioned into one of the three states—*Running*, *Ready*, and *Blocked*. In fact, we assumed that there are three queues of Running, Ready, and Blocked process objects. Being in one of these queues implies that the process object is in a state, the name of which is the same as that of the queue, e.g. when a process object p is in the queue Blocked, it implies that p is blocked and it is waiting for an event to occur. So one process object can be in only one of these queues at a time. Since we have designed Platypus for personal computers with one CPU, just one process can be in the Running queue at a moment.

Note that in a sub-activity diagram, we need to use decision structures whenever we need to strengthen the pre-conditions of an action. In this way, we can make sure that the desired pre-conditions holds when executing the action.

In an activity diagram, the side effects of each activity is expressed using the conjunction of side effects of actions in that activity. For example, the side effects of the activity of Figure 4, is shown in Figure 3.

Finally, it worth mentioning that the pre-conditions of all actions get evaluated in an equal environmental circumstance. In other words, actions in an activity are not executed sequentially, but rather they are executed all together in parallel. It seems that this would degrade the expressiveness of activity diagrams, but it should be noted that the goal of this—the abstract modelling—phase is simply to specify the post-conditions and the desired side effects of each use-case, and we do not want to be involved in the details of the implementation. Bear it in mind that, we make use of sequential property of activity diagrams later in refinement phases.

A use-case diagram is one of the behavioural modelling structures of the UML, which captures functional requirements of the system to be modelled. Use-cases describe the typical interaction between users and the system [2, 13]. The use-cases of the Platypus, consist of system calls, and other functional requirements of the system, such as scheduling. Hence, for each use-case we define a B operation, which has the same name as that of the use-case. We are seeking a way to prove that the side effects of a use-case, which is described in OCL, is

```
post:
if running()->size() > 0 then
 if ready()->size() > 0 then
  let tempSeq = ready@pre->union
   (running@pre) in
    let tempSeqSz = tempSeq->size() in
     ready = tempSeq->subSequence(2,
         tempSeqSz) and
     running = Sequence{ready@pre->
         first()}
 else ...
```

Figure 3: The side effect of the activity shown in Figure 4, specified in OCL.

the equivalent to the side effects of the corresponding B operation. The body of each B operation is described using a single statement, also the side effects of each use-case is described using a single OCL expression. We should find a way to prove the equivalence of an OCL expression with an AMN[2] statement. In the following subsection, we formalise the correspondence condition of two specifications from languages OCL and AMN.

In this subsection, we saw how to model the behaviour of a system using OCL constraints and activity diagrams. In the remainder of this paper, we suppose that we have already modelled all operations on objects—or at least those that have side-effects on system state. When calling each one of the methods (operations) of an object, we consider the activity diagram of that operation as a sub-activity diagram.

2.2 Correspondence of OCL Expressions with AMN Statements

As mentioned earlier, we should examine the correspondence of OCL expressions describing the side-effects of system use-cases, with AMN statements describing corresponding B operations. However, in most cases there is no need to formally prove the equivalence of an OCL expression with a B statement, and usually an informal examination suffices. We can refer to [5, 6] to get a comprehensive list of OCL expressions and their semantics in AMN. We can use the proposed scheme to speed up the process of transformation from OCL to AMN. The following table shows how we can derive AMN expressions from OCL ones.

According to [15], the post-conditions of OCL op-

[2]AMN is the name of the specification language, used in B-Method.

Table 1: OCL expressions, and their semantics in AMN, adopted from [15].

OCL Expression	Semantics in AMN
se : Sequence (T)	$se \in seq(T)$
se1–>union (se2)	$se_1 \frown se_2$
se–>size ()	$size(se)$
se–>subSequence (i, j)	$(se \uparrow j) \downarrow i$
se–>first ()	$first(se)$
st : Set (T)	$st \subseteq T$
st–>forAll(tt|boolexptt)	$\forall(tt).(tt \in st \Rightarrow boolexp_{tt})$
st–>exists(tt|boolexptt)	$\exists(tt).(tt \in st \wedge boolexp_{tt})$

erations are modelled by B substitutions. In this subsection, however, we are proposing a formal approach to examine if an AMN expression (or statement) is equivalent to an OCL expression.

The state of a B abstract machine is constituted by putting the values of all of its variables together at a particular moment. We represent the state of a B abstract machine using a set, associating each variable name to its value; specifically the set $\sigma_M = \{V_1 = v_1, \ldots, V_n = v_n\}$ in a particular moment, means that variable V_1 of machine M has a value of v_1, and so on.

In order to evaluate an OCL expression, we need an evaluation environment which provides access to currently existing objects, and current values of attributes for each object. The exact definition of the evaluation environment is presented in section 2.4. For now, suppose that for each operator in a class, the side-effects of which is described in OCL, we consider two evaluation environments; one pertaining to the state of the system before the execution of the operator, and the other to the state of the system after execution of the operator. These evaluation environments are denoted by γ_{pre} and γ_{post}, respectively.

Definition 1. An *evaluation environment builder* is a function, denoted by $\mathcal{E}$, from the set of current states of a B machine to an evaluation environment for OCL expressions.

Definition 2. The state of a machine M, after the execution of a statement S, is denoted by $\sigma_{M\alpha}[S]$; it is computed by doing all substitutions of the statement in the current state of the machine. Similarly, the state of a machine M before execution of a statement S, refers to the current state of the machine just before executing the statement. State of the machine M before execution of a statement S is denoted by $\sigma_{M\beta}[S]$.

We need an operator to evaluate the OCL expressions according to an evaluation environment. The operator is denoted by $I([\![e]\!])$ for some syntactically well-formed OCL expression e. Generally, the evaluation operator requires the two evaluation environments, γ_{pre} and γ_{post}. We prepare γ_{pre} and γ_{post}, by applying function $\mathcal{E}$ on $\sigma_{M\beta}[S]$ and $\sigma_{M\alpha}[S]$, respectively.

In order to reason about the validity of an OCL-AMN transformation, we take an AMN statement S to be the operation body, the side-effects of which are specified by the OCL expression e.

Definition 3. We say an OCL expression e, specifies an AMN statement S, and denote by $S \equiv e$, if and only if $(\mathcal{E}(\sigma_{M\beta}[S]), \mathcal{E}(\sigma_{M\alpha}[S])) \models I([\![e]\!])$. Meaning that expression e evaluates to *true*, according to evaluation environments $\mathcal{E}(\sigma_{M\beta}[S])$, and $\mathcal{E}(\sigma_{M\alpha}[S])$ that belongs to the state of the system before and after the execution of S.

2.3 Structural Modelling

Among structural modelling constructs of the UML, our method can be applied to transform class diagrams into B models. In general, transforming structural models is easier than transforming behavioural ones. Class diagrams describe data types and relationships among objects in a system. Hence type information and relationships among objects of these types, virtually constitute static structure of the target B abstract machine. In this subsection, we present a roadmap to define the semantics of class diagrams, and develop static structure of B abstract machines according to this semantics. Furthermore, by precisely defining the semantics of UML class diagrams, we can also define an evaluation environment for OCL expressions.

Generally, we consider an object model of a system as a heterogeneous algebra, such that all objects of the system are members of the carrier of the algebra, and operations on these objects reside in the operators set. So an object model, which can be represented using a class diagram, can also be represented with an algebraic system like one shown below:

$$(\{\tau_1, ..., \tau_n\}, \Omega) \tag{1}$$

In this algebra, sub-carriers $\tau_1, ..., \tau_n$ are data types in the model, and each contains all objects of that particular type. Note that the objects in the sets, by themselves, are just identifiers denoting objects in the real world. A point worth mentioning here is that all names in models, including type names, operation names, etc. are members of a set of identifier names that are valid both in OCL and UML; we represent this set by N. In this framework, a type name is in fact a string of characters, however it may be used as a representation of the set of all

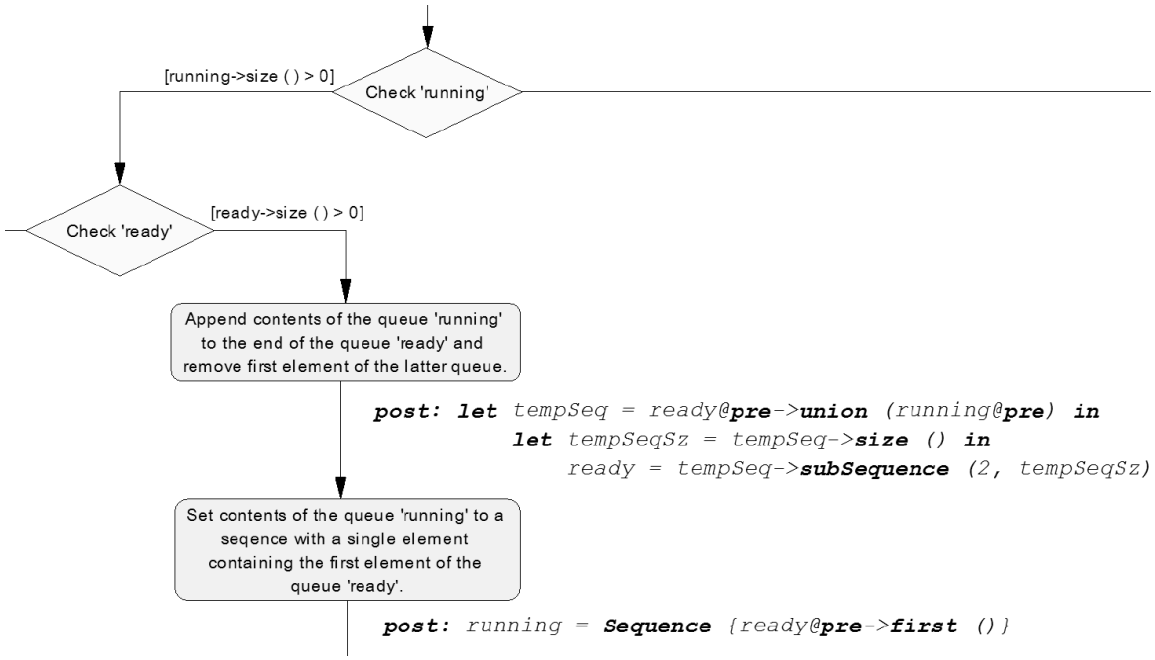

Figure 4: Detailed sub-activity diagram, refining the action Select appropriate process to be dispatched of Figure 2. Note that this diagram is shown partially.

objects of that type. The operator oid is used to explicitly denote the set of all possible objects of a particular type; for a given type name, the operator computes the set of all objects of that type. Each model contains several predefined types, say Integer or String ; other domain specific types can also be defined in each model. More complex types, such as variants, records, and product-types can be constructed on the basis of other types in the algebra. Note that sub-carriers are distinct from each other; relationships such as generalisation would not suggest that a sub-carrier subsumes the other. For example, object 2 in Integer is not identical with that in Real, regardless of the fact that they both represent a single mathematical construct. Further, each sub-carrier has a special value $\perp$, used to express the absence of information of the given type.

The set Ω, is a set of all operators applicable on objects in the system. These operators are of different arity. Each operator with a name ω has the signature:

$$\omega : \tau_i \times ... \times \tau_{i+k} \rightarrow \tau$$

In this formalism, each operator accepts at least an object as input. All of the operators, including those that have side-effects, are described using behavioural modelling constructs.

Each class in an object model is a description of a group of objects sharing the same attributes and behaviour. Hence, each class introduces a set of objects. Each group of objects, having the same characteristics, are considered to be of the type of the class that describes them. In other words, each class is equivalent to a type in the model; this type is a sub-carrier of the mentioned algebra. Each object that can be instantiated on the basis of this class is a member of the sub-carrier, and every operator defined in the class is a member of the operators set. Each operator in Ω is in fact introduced by a class, on the objects of which the operator operates.

When we define the properties[3] of a class of objects, we assign a *visibility* to each property. In this framework, we assume that we have a function named v, which maps each property name in a class to a set of values—namely *public*, *protected*, and *private*. We define this function to have the signature:

$$v : N \times N \rightarrow \{public, protected, private\}$$

As the signature suggests, the function accepts a pair of names (one for class name and the other for property name), and maps the pair to a value in the visibility set.

For each attribute defined in a class, we consider an operator in the set Ω. This operator maps an object of the class, to a value which represents the current value of the attribute. More precisely, each attribute

[3]In this paper, we refer to attributes and operators of a class collectively as properties of the class.

a of type τ in a class C, is equivalent to an operator in the set of operators with the following signature:

$$a : C \to \tau$$

This operator, when applied on an object of type C, returns the current value of the attribute a of the object—which is of type τ. The exact definition of these operators, requires a definition of system state and semantics of OCL expressions. For now, it suffices to say that applying this operator on an object would not change the system state.

Generally, the association relationship describes a connection between the objects of a number of classes. A connection between two objects means that these objects can communicate by sending or receiving messages to/from each other. The connections can be either one-way or two-way. In a one-way connection, a source object can efficiently send messages to a target object, since it has a reference to the target object. In two-way connections, each object has a reference to the other participating object, so both objects can efficiently send messages to each other. Hence an association relationship between two objects means that the source object has a reference to the target object, so we can say that associations are another way of describing attributes of a class, but using associations we can place more information in the model [13]. This information, in fact reflects the information in the real world. By using associations we can express this information easier and more readable. There are special cases of this relationship, which can be used to express whole-part relationship, or describe the relationship between life span of whole-part objects, that bring even more information. But the use of special cases of association relation, say composition and especially aggregation for expressing more information, causes some complexities in transforming the model into B. Furthermore, in most cases, we can develop models just by using associations and compositions [13].

In the proposed method, we assume that UML models to be transformed, have some constructs absent. Of course, we can compensate the absence of these constructs at the expense of models complexity. These restrictions are as follows:

- There is no aggregation relationship. By using association and composition, in most cases, there is no need to use aggregation [13].
- The association is one-way, and target participant of each relationship has a role name. Two-way relationships can be modelled using two one-way relationships.
- The composition relationship is restricted to an association, plus the obligation of creation and destruction of parts by the whole object. In this case, we can model composition using an association, plus a creational/destructive behaviour (using behavioural modelling).

The use of an association or composition, causes B equivalent of our model has several terms to be added to the system invariant. We discuss this issue after introducing system state.

A relationship between two classes is equivalent to an operator that maps each object of the source class to a set (or a sequence) of objects of the target class. For example, assume C1 and C2 to be two classes of a model, and there is an association relationship between them:

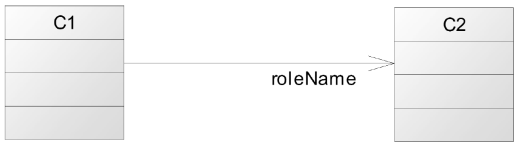

Figure 5: A class diagram with two classes, with an association relationship between them.

There is an operator corresponding to this part of diagram in the set Ω. The signature of the operator is as follows:

$$roleName : C_1 \to S$$

Where, C_1 is the set of all possible objects of the class C1, and S is a set (or a sequence) of objects of the class C2. This is identical to the definition of the attributes of objects—in fact, each association relationship is equivalent to the definition of an attribute in the source class. The type of this attribute can be a set or a sequence of objects of the target class depending on the constraints[4] placed in the target side of the association. The presence of multiplicity in an association has some consequences in the B equivalent of the model. We discuss this issue after introducing system state.

In UML class diagrams, there is a taxonomy relationship other than association and composition; we call this relationship the generalisation [2]. If we represent all type names (sub-carriers names) in the algebra of (1) with a set, say T, the generalisation is a pre-order on T. We denote the relation by a symbol $\prec$. Every pair τ_1 and τ_2 of type names in T contributing in $\tau_1 \prec \tau_2$, means that τ_2 is a parent

[4]By constraint, we mean a constraint such as {ordered}to define a sequence of objects, or other constraints like {unique}, etc.

of (super type of) τ_1. Hence, in each context that an object of type τ_2 is expected, an object of type τ_1 can be safely used. This requires τ_1 to inherit all public (and protected) members of τ_2—i.e. every operator which is applicable on objects of type τ_2, should be applicable also on objects of type τ_1. Bearing the generalisation in mind makes us make a revision in the definition of object properties. To do this, we have to define a couple of concepts.

The set of all parents of a class is the set of all class names that are either immediately or transitively parents of the given class name. We define a function named *parents*, which returns the set of all class names, that are parents of the input class name:

$$parents : N \to \mathbb{P}(N)$$
$$parents(c) = \{c' \mid c \prec c'\}$$

After defining the set of parents of a class, we can define all objects in the hierarchy (of classes). Then we define hoid operator as follows:

$$\mathsf{hoid}(c) = \left(\bigcup_{x \in parents(c)} \mathsf{oid}(x) \right) \cup \mathsf{oid(c)}$$

Note that, the symbol $\cup$ in the function stands for the *untagged union*, which is used to compute the union of sets of objects.

With the presence of generalisation, the signatures of operators corresponding to properties are as follows. For each attribute $a : \tau$, in a class named c, such that for each $x \in parents(c)$, we have $v(x, a) = public$ or $v(x, a) = protected$, then:

$$a : \mathsf{hoid}(c) \to \tau$$

And for each operation with a signature $o(p_1 : \tau_1, \ldots, p_n : \tau_n) : \tau$, in a class named c, such that for each $x \in parents(c)$, we have $v(x, o) = public$ or $v(x, o) = protected$, then:

$$o : \mathsf{hoid}(c) \times \tau_1 \times \ldots \times \tau_n \to \tau$$

Note that a child class can inherit only those attributes and operators that have public or protected visibility in its parent classes, and it is insignificant what kind of visibility do the inherited members have in the child class.

2.4 System State

In subsection 2.2, we realised that we need an environment for evaluating OCL expressions. And in the preceding subsection we noted that the process of translating UML class diagrams requires us to place some constraints on created objects in the system. We define system state in such a way that object creation, destruction, and any change in the properties of each object in the system, would cause a change in the system state. We define system status as follows.

Definition 4. The state of a system which we are modelling, is captured by a tuple (*objects*, *properties*), in which:

- *objects*: Is a function from the set of all type names in the model to a set σ, such that σ is the set of all objects of type of the given type name, created in the system up to a particular moment. More formally, $\sigma \in \mathbb{P}(\mathsf{hoid}(c))$.
- *properties*: Is a function that receives a pair of names, one for class name, say c, and the other for property name. The function returns a function with a signature of $objects(c) \to \tau$, where τ is either a base type (in the case of attributes), or a function type (in the case of operators.) More formally, $properties(c) : N \times N \to objects(c) \to \tau$.

The functions *objects* and *properties* defined in Definition 4, vary over time. For example, suppose that in a system at time t_0, the function *objects* returns σ for a type name say *Process*. At a later time t_1, when the system creates a new object p of type *Process*, the function returns $\sigma \cup \{p\}$—reflecting the creation of the new object in the system. As another example, suppose that in the system of the preceding example, $properties(Process, execution\_time)(p) = 10234$, after $50ms$ we again compute the function application; this time we get 10284, that is the function varies over time.

Now, we can define the evaluation function for syntactically correct OCL expressions. In this paper, we present the semantics of only some simple constructs of the OCL language. We can refer to [8] to get a complete set of rules defining the semantics of the language constructs in terms of a system state similar to that of ours. In the definition of this function, we assume that we are going to consider single state of the system. The generalisation of this function for two system states is straightforward; we should evaluate sub-expressions in a composite expression in two different environments. We assume further that besides system state, we have a variable-mapping function that maps each variable name to its value; we denote this function by β. We call a pair of system status and a variable-mapping function, an evaluation environment, and denote by γ. We define rules, proving

judgements of the form $I([\![e]\!])$, inductively as follows:

$$\frac{}{I([\![\gamma]\!]) = \beta(v)} \tag{2}$$

$$\frac{\gamma' = ((objects, properties), [v \mapsto I([\![e_1]\!](\gamma))]\,\beta)}{I([\![let\ v = e_1\ in\ e]\!])(\gamma) = I([\![e]\!])(\gamma')e} \tag{3}$$

$$\frac{}{I([\![undefined]\!])(\gamma) = \bot \in OclAny} \tag{4}$$

$$\frac{\begin{array}{c}\left(\begin{array}{c} a_1 = I([\![e_1]\!])(\gamma)\ if\ I([\![e_1]\!])(\gamma) \in \tau_1 \vee \\ a_1 = \bot \in \tau_1\ if\ I([\![e_1]\!])(\gamma) = \bot \in OclAny \end{array}\right) \\ \vdots \\ \left(\begin{array}{c} a_n = I([\![e_n]\!])(\gamma)\ if\ I([\![e_n]\!])(\gamma) \in \tau_n \vee \\ a_n = \bot \in \tau_n\ if\ I([\![e_n]\!])(\gamma) = \bot \in OclAny \end{array}\right) \\ properties(c, \omega) : \tau_1 \times \ldots \times \tau_n \to \tau \\ for\ some\ \tau_1, \ldots \tau_n, \tau\ and\ c \end{array}}{\begin{array}{c} I([\![\omega(e_1, \ldots, e_n)]\!])(\gamma) = \\ properties(c, \omega)(a_1, \ldots, a_n) \end{array}} \tag{5}$$

$$\frac{}{\begin{array}{c} I([\![if\ e\ then\ e_1\ else\ e_2]\!])(\gamma) = \\ \left\{\begin{array}{ll} I([\![e_1]\!])(\gamma) & ; I([\![e]\!])(\gamma) = true \\ I([\![e_2]\!])(\gamma) & ; I([\![e]\!])(\gamma) = false \\ \bot & ; otherwise \end{array}\right. \end{array}} \tag{6}$$

Note that this function, if defined completely, maps each well-formed OCL expression to a mathematical construct, i.e. defines the denotational semantics of the language.

By the use of system state, we can also define constraints that are needed when we are translating class diagrams into AMN. These constraints are needed when we use constructs such as composition and multiplicities in association relationships. In the preceding subsection, we introduced composition relationship to be a relationship between a whole object and its parts. We assumed a restriction on this relationship, to be a simple association plus a behaviour that the whole object is responsible for creation and destruction of its parts. This is accomplished simply by creating parts when a whole is created, and conversely by destructing parts when their whole is destructed. Note that the obligation of objects creation and destruction depends on the implementation language, for example in a language like C++ each object has a constructor and a destructor. The behaviour of these operators models the composition construct—i.e. the whole object in its constructor creates part objects and destructs the parts in its destructor.

By means of multiplicities in associations, we can state that how many objects can be connected together through an association [2]. When we state a multiplicity n, at one end of an association, we are specifying the fact that for each object of the class at the opposite end there must be exactly n objects of the class at the near end [16]. In the preceding subsection, we examined the case in which the multiplicity is placed at the target side of an association. Now we are examining the case where the multiplicity is put on the source side of the association. Consider the following figure that represents a part of a class diagram.

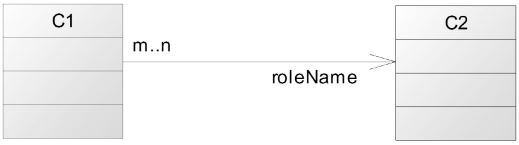

Figure 6: Two classes connected to each other using an association, with a multiplicity constraints.

In the above diagram, we have the following constraint. Each object of type C2 can receive messages from at least m objects, and at most n objects of type C1. In other words, the number of objects of type C1 that we can apply an operator on them and get an object of type C2, is restricted to the naturals numbers between m and n. We can place such constraints in terms of objects properties, but defining constraints using *properties* function complicates system invariant, and hence proving system consistency gets harder. If we need to verify the fact that our model satisfies this kind of constraints, we need to use other approaches to model association relationships, as described in [8].

2.5 Mapping the Algebra to a B Model

In the preceding subsection, we noted that using an algebraic system and several auxiliary functions, we can formally restate the UML models that are stated pictorially. In this subsection, we present an algorithm to map the precise representation of object models to a B model. The algorithm builds up a part of static structure of the B model, and the remainder of its static structure is built by another algorithm presented next. Using these algorithms we can model system state besides the definition of data types. In these algorithms, we assume that the resulting B abstract machine is stored in a data structure, in a way that we can add terms in its clauses, or else search for the terms that have already been added to some clause of the machine. Our algorithms add terms to the data structure or searches for existing terms, using an Abstract Data Type (ADT). The ADT is presented below:

StringList in the above figure, is the data type for lists of string. In this data structure, each list stores

```
package     AbstractMachineADT {
            ...
            clauseSETS : StringList;
            clauseCONSTANTS : StringList;
            clausePROPERTIES : StringList;
            ...
interface:
            add (term : String, clause
                : {SETS, CONSTANTS, ...}) : Unit;
            inClause (clause :
                {SETS, CONSTANTS, ...},
                term : String) : Boolean;
}
```

Figure 7: A code snippet describing the ADT managing a B abstract machine information.

terms of the corresponding clause of the abstract machine, e.g. the set $clauseSETS$ stores terms of the $SETS$ clause of the resulting machine. The operator *add*, adds a term that it has received by its first parameter to the list of terms in a clause of the abstract machine corresponding to the literal that it has received by its second parameter. We assume that the operator chooses appropriate connective between terms depending on the kind of clause of the machine. The return type of the add operator is a type with just a single element, this element announces the client that the computation has been terminated. The operator *inClause* is used to search a term in the clause *clause* of the machine. If the term is found in the list then the operator return $true$, else it returns $false$.

After the necessary parts of the ADT are defined, we now present the algorithm of transformation of the algebra, equivalent to a UML model, into a B model. In Figure 8, we present the algorithm.

The input for this algorithm is a heterogeneous algebra, and its output is a B abstract machine. Line 1 of the algorithm, adds a deferred set named ran_oid in the $SETS$ clause of the resulting machine. In subsection 2.3, we introduced an operator named oid, we defined that this operator to receive a class name, and return the set of all object identifiers, that we can instantiate on basis of the given class. The set ran_oid, in fact is defined to be the range of the oid operator. Hence, the set contains all possible object identifiers that we can instantiate on basis of each class in the source model. As before, each object identifier denotes an object in the real world, but since in this phase of modelling we do not need to specify the exact characteristics of the object identifiers, we define the set as a deferred set.

In lines $2-a$ and $2-b$, we add each type name in the model as a constant in the $CONSTANTS$ clause of the resulting machine, and define its type

```
algorithm   TransformAlgebraIntoB of
input       an algebra
output      result : AbstractMachineADT
is {
            1. result.add(ran_oid, SETS);
            2. for each type name τ in the algebra do
               a. result.add(τ, CONSTANTS);
               b. result.add(τ ∈ ℙ1(ran_oid),
                                 PROPERTIES);
               c. result.add(null_τ, CONSTANTS);
               d. result.add(null_τ ∈ τ,
                                 PROPERTIES);
               e. result.add(hoidτ, CONSTANTS);
               f. result.add(hoidτ ∈ ℙ1(ran_oid),
                                 PROPERTIES);
            3. if result.inClause(CONSTANTS, τ1)∧
               result.inClause(CONSTANTS, hoidτ1)∧
                  ⋮
               result.inClause(CONSTANTS, τn)∧
               result.inClause(CONSTANTS, hoidτn)
               then
               a. result.add(τ1 ∩ ... ∩ τn = {},
                                 PROPERTIES);
               b. result.add(τ1 ∪ ... ∪ τn = ran_oid,
                                 PROPERTIES);
            4. for each type name τ in the algebra
                  such that there exists at least one
                  type name τ' such that τ ≺ τ' do
            ▷ Note that there must be no τ'' such
              that τ ≺ τ'' ≺ τ'
               a. result.add(hoidτ = τ ∪ hoidτ',
                                 PROPERTIES);
            5. for each type name τ in the algebra
                  such that there is not exist any
                  type name τ' such that τ ≺ τ' do
               a. result.add(hoidτ = τ,
                                 PROPERTIES);
            6. return result;
}
```

Figure 8: The algorithm for transforming an input object model, represented by a heterogeneous algebra, into a B model.

to be a non-empty subset of the ran_oid set. In a similar manner, in lines $2-e$ and $2-f$, for each type name τ in the model we add a constant named $hoid\tau$, and define its type to be a subset of object identifiers. Note that, by putting terms side-by-side we mean a string concatenation. For example, taking τ be a string, and *hoid* another string value, the term $hoid\tau$ concatenates the two values. Note also that, these sets are defined to be non-empty, since at least a special *null* value would exist in each one (as noted before in subsection 2.3). In lines $2-c$ and $2-d$, for each type name in the model we add a constant representing a null value of that type.

In lines 3, $3-a$, and $3-b$, for all constant names of the form $\tau_1, \ldots, \tau_n$ (except those constant names of the form $hoid\tau_1, \ldots, hoid\tau_n$), that exist in the resulting machine, we add the predicate $\tau_1 \cap \ldots \cap \tau_n =$

$\{\} \wedge \tau_1 \cup \ldots \cup \tau_n = ran\_oid$ in its PROPERTIES clause. This means that the set ran_oid is partitioned into subsets $\tau_1, \ldots, \tau_n$.
In line $4-a$ of the algorithm, for each type name τ in the model, which has an immediate parent τ' we add a term of the form $hoid\tau = \tau \cup hoid\tau'$. Next in line $5-a$ of the algorithm, we define *hoid* for those type names that have not any parent to be equal with the given type itself. Since the set *parents* for these type names is empty. These two lines, iteratively define the *hoid* of all type names in the model. The last line of the algorithm returns the resulting abstract machine to the client.

2.5.1 Building up the Remainder of the Static Structure

In the beginning of this subsection, we presented an algorithm for transforming a heterogeneous algebra, representing an object model, into a B abstract machine. The algorithm, in fact builds those parts of the static structure of the B machine, that define the types in the model. Now, we present another algorithm for building the remainder of the static structure of the resulting machine. The ADT used in this algorithm is the identical ADT shown in Figure 7. However, we need to reveal two more data fields of it. These fields are shown below.

```
package     AbstractMachineADT {
            ...
            clauseVARIABLES : StringList;
            clauseINVARIANT : StringList;
            ...
interface:
            ...
}
```

Figure 9: The ADT of Figure 7, with two more fields revealed.

The algorithm for building up the remainder parts of the static structure of the abstract machine is shown in Figure 10. The input to the algorithm is a heterogeneous algebra, and the output of the first algorithm. The output of the algorithm is a B abstract machine, which all of the static information in the UML model is reflected in it. In line 1 of the algorithm, the resulting abstract machine is initialised with the input machine; since in this algorithm we add some more specifications to an existing machine. In lines $2-a$ and $2-b$ of the algorithm, for each type name τ in the input machine (except for those of the form $hoid\tau$), we add a variable named $sigma\tau$ in the resulting machine. This variable is defined to be a subset of type τ. At any instant of time, this variable holds all object of type τ, that are created

```
algorithm   BuildSystemState of
input       an algebra,
            inputMachine : AbstractMachineADT
output      result : AbstractMachineADT
is {
            1. result ← inputMachine
            2. while inputMachine.inClause
                  (PROPERTIES, τ ∈ ℙ1(ran_oid)) do
               a. result.add(sigmaτ, VARIABLES);
               b. result.add(sigmaτ ∈ ℙ(τ),
                                 INVARIANT);
               c. for each operator a : τ → τ'
                  in the set Ω of the algebra
                  such that τ' is not a function type do
                     i. result.add(τ_a, VARIABLES);
                     ii. result.add(τ_a ∈ sigmaτ → sigmaτ',
                                 INVARIANT);
}
```

Figure 10: The algorithm for building the state of the system.

until that moment. This variable, in fact is equal with $objects(\tau)$. In line $2-c$, for each operator a, applicable on objects of type τ, which returns an object of another type τ', we add a variable to the resulting machine. Note that, the return type of the operator should not be function type, i.e. a should not be an operator defined in the class τ. This variable is a total function from the set $sigma\tau$ to the set $sigma\tau'$. The function has to be *total*, since attributes of a class are defined for all objects of the class.

3 The Development Process

In the preceding section, we proposed a method for translating UML models into B models. Using this tool, we are able to formally verify our models. Once we have verified the model in a particular level of abstraction, we can add further details to the model (or, interchangeably in writing, refine the model). The development process that we propose, is an iterative one, and can be used in elaboration phase of the RUP. We refine the UML model at each iteration, and add further implementation details to the model. The activities of the process is depicted in Figure 11.
According to Figure 11, a UML model in a particular level of abstraction is translated into a B abstract (, or possibly into a B refinement) machine. We then verify the correctness (consistency) of the B model; once the consistency is proved, we claim that we have a verified UML model, or else we have to revise the UML model. When we have proved the consistency of the model we can either stop the process, or

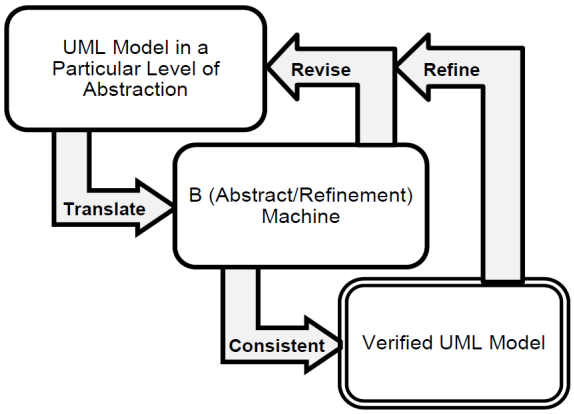

Figure 11: The stages of the process, and artefacts of each stage.

further refine the model. When we add implementation details to the model, the model becomes more concrete and hence we can better reason about the performance of the system to be implemented.
In B-Method, a refinement machine must describe the relationship between its own state and the state of the machine it refines [3]. We call this description the *linking invariant*. Since, in UML, there is no way of relating two models, e.g. a model refines or implements the other [9], our proposed method as a result also lacks the ability to add linking invariants in the refinement machines. This is the responsibility of the system engineer to add these linking invariants in each refinement (or implementation) machine manually.

3.1 Removing Irrelevant Information

As we noted in section 2, the proposed method introduces a lot of irrelevant information into a B machine. We have to remove the information from the machine before implementing the machine, since book keeping of objects of the system is the responsibility of the memory manager (of runtime system, or memory management sub-system embedded in the system itself). Therefore data structures storing the information should not be implemented. But as we saw in section 2, the proposed method does not let us not to add the objects book keeping information to refinement machines. In fact, this the responsibility of the engineer to remove the information when the model is refined to an extent. To remove irrelevant information from refinement machine, we can move the information to separate (independent) abstract machines, and later communicate with these machines through their interface. It worth mentioning that the removal of information is done only for once, since when the structural model contains sufficient information and details, we do not need to refine it any more. Henceforth, we refine the actions of activity diagrams; these actions are written in terms of the interface of machines managing objects of each type. Generally, the interface of these machines should contain the following operations:

- An operation for constructing new objects. This operation may accept extra information in order to initialise the attributes of the newly created object.
- Accessor operations, for accessing the current value of attributes for each existing object.
- Mutator operations, for changing the current value of attributes for each existing object.
- An operation for destructing existing objects. This operation returns a given object to the pool of free object identifiers of the system.

We do not need to refine and implement these machines, since primitive constructs of object-oriented languages enable the programmer to describe these operations. Having a verified compiler (and a runtime system, if exists any), we can assume that these operations are verified, and hence using them would not make our system inconsistent (from memory management and objects book keeping stand point of view, of course). Once we moved irrelevant information to separate auxiliary machines, we can verify them separately and only for once; thereby letting us concentrate on proof obligations related to the functional aspects of the system. Furthermore, the efforts for adding linking invariants in refinement machines is also restricted to those data structures that are considered to be implemented.
When we refine activity diagrams, we can reach a level of details such that we can easily express the actions in a diagram using instructions (or group of instructions) of a host machine. Hence, we can do almost exact approximations on the performance efficiency of the system to be implemented, or we can design the system in such a way that the system performs efficiently on the host machine.

4 More On *Platypus*

Platypus is an operating system kernel, which is developed using the method proposed in this paper. This kernel is developed just for the sake of showing that the proposed method is practical, and otherwise it is not an operational OS kernel at all.

Table 2: System calls provided by the Platypus kernel.

createProcess	
No.	0
Int.	0x20
Description	Creates a child process.
forceSchedule	
No.	1
Int.	0x20
Description	Removes a process from blocked processes list, and appends it to the ready queue.
abortProcess	
No.	2
Int.	0x20
Description	Halts the execution of a process.
getPID	
No.	3
Int.	0x20
Description	Retrieves the process ID of caller process and its parent, pager, and exception manager processes.
mapPage	
No.	0
Int.	0x21
Description	Makes a page of a process sharable among other processes.
grantPage	
No.	1
Int.	0x21
Description	Removes a page of a process and adds it to the address space of some process.
reclaimPage	
No.	2
Int.	0x21
Description	Revokes all mapped pages.
sendMessage	
No.	0
Int.	0x22
Description	A non-blocking system call for sending a message (exactly 24 bytes in size) to some process.
receiveMessage	
No.	1
Int.	0x22
Description	A blocking system call for receiving a message from a specific or any process.

Because we have not done any experiment on the performance efficiency of it, and even any throughout examination on the completeness of its system calls. This kernel is designed using UML for IA-32 architectures, and is implemented in approximately 1.3 *kLOC* of C++. At any stage of development, we translated the designed UML model into an equivalent B model, and its correctness and also its consistency with its more abstract version got proved. As mentioned earlier, refinement is done in UML and B method is used only for formal proof of correctness and consistency in refinements. In the case of the Platypus, the structural model was precise enough in the first iteration; hence we did not refine it any more. We refined behavioural model two more stages, on the other hand. However, we proved the correctness of second and third refinements and their equivalence with each other and with the abstract model, it would be nice to prove the correctness of the implementation too[5]. Platypus provides 9 system calls overall. We categorise these system calls into three classes as follows:

- Calls for process management.
- Calls for memory management.
- Calls for inter-process communication.

We set up three service routines for interrupts `0x20`, `0x21`, and `0x22` to provide these system calls. When doing system call, the caller process should place system call number in register `EAX`, and the remainder arguments in registers `EBX`, `ECX`, `EDX`, `ESI`, and `EDI`, respectively. System calls do not return any result to the caller process. We illustrate the list of system calls, and a concise description of each one in Table 2.

In this system, for each process as it is created, a parent, a pager, and an exception manager process is assigned. Parent is the creator process and has enough privilege to abort, or to reschedule any of its child processes. Pager, is a process that handles page faults of its clients. Any page fault occurred in a process, causes the pager associated with that process receive a message containing enough information to handle the page fault; the pager then can map or grant one of its pages to the client process. Pagers are privileged to abort their clients when they cannot service. In this version of the Platypus, there is only one pager, with process ID of 1. Exception manager, is a process that handles exceptions of its clients. The process of handling exceptions is just like handling page faults. In this version of the Platypus, there is only one exception manager process with process ID of 1. The process with ID 1 is also the parent of all processes in the system and it is automatically loaded in the boot process.

5 Conclusion

We started project Platypus with an aim to increase the productivity of formally verified microkernels development. In this paper, we explained the applied

[5]It could be done using rules mapping AMN statements to C++ statements, according to their semantics.

methods and the development process of the system. Our main idea is to use the UML in conjunction with formal modelling techniques, especially with the B-Method. In section 1, we stated the importance of the project and explored related works. In section 2, we explained our method for transforming UML behavioural models into B models; we also developed formal semantics of UML structural models, and proposed algorithms to translate these constructs into a B model. In section 3, we introduced a development process to use our proposed methods to develop systems. And finally in section 4, we took a bird's eye view of the Platypus.
The development team of the Platypus, comprises two software engineers. It took the team less than five months to design, formally verify, and implement the system.

References

[1] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an OS kernel," in *SOSP*, pp. 207–220, ACM, 2009.

[2] B. G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*. Addison Wesley, 2005.

[3] J. Abrial, *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.

[4] C. F. Snook and M. J. Butler, "UML-B: Formal modeling and design aided by UML," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 1, pp. 92–122, 2006.

[5] H. Ledang and J. Souquières, "Integrating UML and B Specification Techniques," in *GI Jahrestagung (1)*, pp. 641–648, 2001.

[6] H. Ledang and J. Souquières, "Modeling Class Operations in B: Application to UML Behavioral Diagrams," in *ASE*, pp. 289–296, IEEE Computer Society, 2001.

[7] L. Shan and H. Zhu, "A Formal Descriptive Semantics of UML," in *ICFEM*, vol. 5256 of *Lecture Notes in Computer Science*, pp. 375–396, Springer, 2008.

[8] *Object Management Group (OMG). OMG OCL 2.0 Specification, 2003.*

[9] D. Bjorner, *Software Engineering 2: Specification of Systems and Languages*. EATCS Series, Springer, 2006.

[10] J. Liedtke, "On micro-kernel construction," in *SOSP*, pp. 237–250, 1995.

[11] W. Stallings, *Operating Systems: Internals and Design Principles*. Prentice Hall, 2011.

[12] *Clearsy. Atelier B reference manual, 2012.*

[13] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 3 ed., 2003.

[14] B. Meyer, "On formalism in specifications," *IEEE Software*, vol. 2, no. 1, pp. 6–26, 1985.

[15] H. Ledang and J. Souquières, "Derivation schemes from ocl expressions to b," tech. rep., LORIA, 2002.

[16] J. Whitten and L. Bentley, *Systems analysis & design methods*. McGraw-Hill/Irwin, 2007.