# A Case Study on the Lightweight Verification of a Multi-Threaded Task Server ☆

Néstor Cataño[a,*], Ijaz Ahmed[b], Radu I. Siminiceanu[c], Jonathan Aldrich[d]

[a] *Carnegie Mellon University - Portugal, The University of Madeira*
*Campus da Penteada, Funchal, Portugal*
[b] *Carnegie Mellon University - Portugal, Madeira ITI*
*Campus da Penteada, Funchal, Portugal*
[c] *National Institute of Aerospace, Hampton VA, USA*
[d] *Institute for Software Research, School of Computer Science*
*Carnegie Mellon University, Pittsburgh PA, USA*

**Abstract**

We present a case study of verifying the design of a commercial multi-threaded task server (MTTS), developed by the Novabase company, used for massively parallelising computational tasks. In a first stage, we employed the Plural tool, which is designed to perform lightweight verification of Java programs using a Data Flow Analysis (DFA) framework, to specify and verify the MTTS. We wrote the Plural specification for the MTTS based on the code developed by Novabase, its informal documentation, and our discussions with Novabase engineers, who validated our understanding of the MTTS application. The Plural specification language is based on *typestates* and *access permissions*. In a second stage, we developed the Pulse tool that enhances the analysis performed by Plural, and used the tool on the MTTS specifications. Pulse translates Plural specifications into an abstract state-machine model that captures the semantics of all the possible concurrent programs implementing the given specifications, and uses the `evmdd-smc` symbolic model-checker to verify the machine model. The experimental results on the MTTS specification show that the exhaustive model-checking approach scales reasonably well and is efficient at finding errors in specifications that were not previously detected with the Data Flow Analysis (DFA) capabilities of Plural.

*Keywords:*
Concurrency, Parallelism, Formal Methods, Specification, Verification, Model-Checking, Program Analysis.

☆This work has been supported by the Portuguese Research Agency FCT through the CMU-Portugal program, R&D Project Aeminium, CMU-PT/SE/0038/2008.

*Corresponding author

*Email addresses:* `ncatano@uma.pt` (Néstor Cataño), `ijaz.ahmed@m-iti.org` (Ijaz Ahmed), `radu@nianet.org` (Radu I. Siminiceanu), `jonathan.aldrich@cs.cmu.edu` (Jonathan Aldrich)

## 1. Introduction

Multi-core processor platforms are poised to become massively parallel within the next few years. To take advantage of this new technology, computer scientists are working on the development of programming languages and programming paradigms that exploit the parallel computing power provided by the new hardware, while at the same time helping programmers ensure correctness in a parallel context.

Developing correct parallel programs is difficult, in part because of the danger of interference betwen tasks. If two tasks read and write to shared, mutable state, they must be synchronized to ensure that the state of the program does not become inconsistent. Unfortunately, it is easy for programmers to omit synchronization or to incorrectly synchronize, resulting in errors such as race conditions when multiple threads read and write to the same locations in the heap.

In order to help programmers reason about access to shared mutable state, researchers have therefore developed a number of abstractions that can be used to characterize the way that multiple threads access potentially shared state. An early example of this was Boyland et al.'s *capabilities for sharing*, which are annotations that can be associated with references in the program [1]. A capability (now more commonly called a *permission* [2]) restricts the way that different threads can use the object to which the reference points. For example, a **Unique** permission describes an object that is writable but not shared between threads, while an **Immutable** permission describes an object that is potentially shared between threads, but cannot be written to. In either case, the object can be accessed following the rules of the permission without fear of introducing a race condition.

Permissions and related concepts have since been applied to address issues related to safe concurrency [3] as well as security [4]. This paper focuses on a case study about the use of (the existing tool) Plural [5] to specify and verify concurrency properties of an industrial application, and on our development of the Pulse tool [6] that enhances the analysis performed by Plural. The Plural specification language is based on a form of permissions known as *access permissions* [7] as well as *typestates*. While access permissions describe whether an object can be read or written through various references, typestates specify the legal order of operations that may be applied to the object, specified as a state machine [8]. The two forms of specification are complimentary: in addition to helping to verify safe concurrency, permissions enable a dataflow analysis to more precisely track the changes to the typestate of an object as the program executes.

We used Plural for the specification and verification of a multi-threaded task server (MTTS), implemented by Novabase, which has been extensively used for massively parallelising the processing of computational tasks. MTTS is part of several software applications used by Novabase's clientele, e.g. it has been used in the financial sector to parallelise the archiving and processing of documents. The MTTS application is implemented in Java, and uses queues to store tasks that are executed by a pool of threads.

The MTTS case study revealed a series of limitations of the analysis performed by the Plural tool. These limitations are related to the impossibility of exhaustively analysing arbitrary sequences of method calls and with checking Plural specifications

in isolation. We therefore developed the Pulse tool, as an Eclipse plug-in [6], to overcome these limitations. Pulse translates Plural specifications into abstract state-machine models that capture the semantics of all the possible concurrent programs implementing a given specification. We used the `evmdd-smc` [9] symbolic model-checker, a minimalist high-performance tool with an input language inspired by SAL (Symbolic Analysis Laboratory) [10], to model the abstract machines and to perform the verification.

The contributions of Pulse to the analysis performed by Plural are: (*i*.) an exhaustive yet tractable approach to analyse specifications at an appropriate level of abstraction for concurrent execution; (*ii*.) a discrete semantics for access permission manipulation that is scalable; (*iii*.) a translation algorithm that allows full automation; (*iv*.) an integration of model checking techniques into the Plural framework. The verification methodology implemented in Pulse is not limited to Plural specifications alone, but may be employed for any language that captures the concepts of typestates, permissions and concurrency, such as the Fugue checker [11].

The rest of this paper is organised as follows. Section 2 introduces the Plural and Pulse tools. Section 3 presents our specification of the MTTS application, shows miscellaneous aspects of the verification of the MTTS with Plural, and discusses about the limitations of the analysis performed by Plural. This discussion includes a list of desirable features regarding the analysis done by Plural. Sections 4 and 5 lay the theoretical background for the automated extraction of discrete state models from Plural specifications that is implemented in the core of Pulse. Section 6 presents the results of using Pulse on the specifications for the MTTS application. Section 7 presents related work, and Section 8 discusses future work and conclusions.

## 2. Preliminaries

### *2.1. The Plural Tool*

Plural is a specification and checker tool, implemented as a plug-in to Eclipse [5]. The Plural tool takes a Java program annotated with Plural specifications and checks whether the program complies with its specifications or not. Plural implements a simple effects analyser that checks if a particular method has side effects, and an annotation analysis tool that checks whether annotations are well formed.

#### *2.1.1. The Plural Specification Language*

The Plural specification language is a general language designed to facilitate the development of component-based and concurrent software. The Plural specification language combines *access permissions* and *typestates* specifications. Access permissions are abstract capabilities allowing a method to access a particular object state [1]. Plural uses access permissions to keep track of the various references to a particular object, and to check the types of accesses these references have. Accesses can be reading or writing (modifying). Typestates define protocols on finite state machines [8]. They describe the sets of valid object states on which a method can be called.

| This reference | Other references |
|---|---|
| **Unique** | $\emptyset$ |
| **Full** | **Pure** |
| **Share** | **Share**, **Pure** |
| **Pure** | **Full**, **Share**, **Pure**, **Immutable** |
| **Immutable** | **Pure**, **Immutable** |

| Current permission | | Access through |
|---|---|---|
| read/write | read-only | other permission |
| **Unique** | - | none |
| **Full** | **Immutable** | read-only |
| **Share** | **Pure** | read/write |

Figure 1: Simultaneous access permissions taxonomy [7]

Plural provides five types of access permissions, namely, **Unique**, **Full**, **Share**, **Pure**, and **Immutable**. Figure 1 presents a taxonomy of how different access permissions can coexist, e.g. **Full** access to a referenced object allows the existence of any other reference with **Pure** access to the same referenced object.

- **Unique(x)**. It guarantees that reference **x** is the sole reference to the referenced object. No other reference exists, so **x** has exclusive reading and modifying (writing) access to the object.
- **Full(x)**. It provides reference **x** with reading and modifying access to the referenced object. Additionally, it allows other references to the object (called aliases) to exist and to read from it, but not to modify it.
- **Share(x)**. Its definition is similar to the definition of **Full(x)**, except that other references to the object can further modify it.
- **Pure(x)**. It provides reference **x** with read-only access to the referenced object. It further allows the existence of other references to the same object with read-only access or read-and-modify access.
- **Immutable(x)**. It provides **x** and any other existing reference to the same referenced object with non-modifying access (read-only) to the referenced object. An **Immutable** permission guarantees that all other existing references to the referenced object are also read-only **Pure(x)** or **Immutable(x)** permissions.

Plural specifications are embedded in Java code within special marked comments. A simple Plural specification for a method combines pre- and post-conditions, embedded immediately before the method declaration. The pre-condition describes (1) the typestate the object must be before the method starts, and (2) the type of access the object permits. In the spirit of Girard's Linear Logic [12], access permissions are

produced and consumed. The method post-condition describes the produced access permissions and the typestate the object will be after the method ends.

In Plural, a method specification is written with the aid of a **@Perm** clause[1], composed of a **requires** part, describing the resources required by the method to be executed (the pre-condition), and an **ensures** part (the post-condition), describing the resources generated after method execution. Some methods can legally be called with and can produce different sets of resources (declared within a **@Cases** clause). Hence, if the client does not know which case pre-condition in the specification will be selected, it must be prepared to deal with any of the post-conditions listed by the **@Cases** specification. A typestate is declared within a **@State** clause, and several typestates are made available inside a **@ClassStates** declaration. Additionally, an object can be in different *dimensions* representing the dynamic typestates of the object. In Plural, dimensions are declared with the aid of the **@dim** keyword.

Figure 2 illustrates an example of a Plural specification taken from the MTTS case study presented in Section 3. Class Task models a generic task in the MTTS. The internal information about the task is stored in an object "data" of type MttsTaskDataX. A task can be in four possible typestates: **Created**, **Ready**, **Running**, or **Finished**. The constructor of class Task allocates a **Unique** object that is in state **Created**. A task is in state **Ready** once it has been given some data to be executed. It is in state **Running** when it is running, and it is in state **Finished** when it has been executed and its data has been consumed.

Looking at the state transitions in more detail, method "setData()" requires **this** to have **Full** permission to its referenced object, which should be in state **Created**, and simultaneously requires that its first parameter is different than **null**. The operator "*****" combines several specifications. Method "execute()" requires **this** to have **Full** permission and to be in state **Ready**, and ensures that **this** will have **Full** permission on its referenced object, which will be in state **Finished**.

### *2.2. The Pulse Tool*

We implemented Pulse as an Eclipse plug-in that works on top of the Plural tool. The Pulse tool [6] takes a Plural annotated Java spec and produces an abstract state machine model expressed in the input language of the `evmdd-smc` [9] model-checker[2]. We use the state-of-the-art `evmdd-smc` symbolic model checker [3] to verify that the Plural specifications satisfy a set of basic integrity properties. The input language of `evmdd-smc` is similar to SAL (Symbolic Analysis Laboratory), however `evmdd-smc` is more efficient than SAL for several reasons. The `evmdd-smc` is powered by an edged-valued decision diagrams (EVMDD) library, `libevmdd`[3] that can be orders of magnitude faster [9] than the ubiquitous CUDD, especially for models that capture concurrency. Secondly, the much leaner `evmdd-smc` is free of all of the syntactic sugar provided by SAL, which often leads to tremendous pre-processing overhead.

[1] Alternatively, Plural allows the use of **@Case**.

[2] Sections 4 and 5 give full details on how these models are generated.

[3] Available at `http://research.nianet.org/~radu/evmdd/`.

```
@ClassStates({
 @State(name = ``Created'', inv = ``data == null''),
 @State(name = ``Ready'', inv = ``data != null''),
 @State(name = ``Running'', inv = ``data != null''),
 @State(name = ``Finished'', inv = ``data == null'')
})
public Task implements AbstractTask {
  private MttsTaskDataX data;

  @Perm(ensures = ``Unique(this) in Created'')
  public Task() { ... }

  @Perm(requires = ``Full(this) in Created * data != null'',
        ensures = ``Full(this) in Ready'')
  public void setData(MttsTaskDataX data) { ... }

  @Perm(requires = ``Full(this) in Ready'',
        ensures = ``Full(this) in Finished'')
  public void execute() throws Exception { ... }
}
```

Figure 2: Example of a specification for a generic task

The ability to express custom temporal logic properties for concurrent programs gives `evmdd-smc` further freedom to perform verification tasks tailored to each application.

In the following, we explain the different `evmdd` sections generated for the Plural specification in Figure 3. The figure introduces the MttsTask class that models a generic programming task. The internal information about the task attributes is stored in an object data of type MttsTaskDataX. An object of type MttsTaskDataX can be (in typestate) **Empty** or **Filled**. The state **Empty** represents the fact that the data has not been set and state **Filled** indicates that data has been set. An object of type MttsTask can be in state **Created**, **Ready**, **Complete**, or **Destroyed**. It is in state **Ready** once it has been given some data to be run (by method setData). It is in state **Complete** (method execute) when its execution has been finished, and it is in state **Destroyed** (method delete) when it does not exist any more. Method setData requires the object **this** to have **Full** access to its referenced object, to be in state **Ready**, and requires the parameter "d" to have **Pure** access to its referenced object. Method setData further ensures that on method termination, the object **this** will be in state **Filled** with **Pure** access permission. Method getTaskStatus requires **Pure** access permission and provides the status of the task such as ready, running etc.

a) **Declarations/Initialisations.**

The model declares eight variables to represent the typestate (`state`), the class method (`meth`), the total number of read (`tkrB`) and write (`tkwB`) tokens, the program counter (`pc`), the total number of access permissions (`ap`), and the number of read (`tkr`) and write (`tkw`) an object reference owns. The numbers between brackets are the possible values a variable can have, where `0` is used in all cases to represent the value "undefined". All the initial state values are undefined except for the total number of read and write tokens.

```
@ClassStates({
 @State(name="Created",   inv="data!=null"),
 @State(name="Ready",     inv="data!=null"),
 @State(name="Complete",  inv="data!=null"),
 @State(name="Destroyed", inv="data == null")
 })
public class MttsTask {

private MttsTaskDataX data;

@Perm (ensures= "Unique(this) in Created")
MttsTask() { ... }

@Perm(requires="Full(this) in Created*Pure(d) in Filled",
      ensures="Full(this) in Ready")
public void setData(MttsTaskDataX d) { ... }

@Cases({
@Perm(requires="Pure(this) in Ready", ensures="Pure(this) in Ready")
@Perm(requires="Pure(this) in Complete",ensures="Pure(this) in Complete")
})
@Perm(ensures="Pure(result) in Filled")
public MttsTaskDataX getData() { ... }

@Perm(requires="Full(this) in Ready", ensures="Full(this) in Complete")
public void execute() { ... }

@Perm(requires="Full(this) in Complete", ensures="Full(this) in Destroyed")
public void delete() { ... }

@Perm(requires="Pure(this)", ensures="Pure(this)")
public int getTaskStatus() { ... }
```

Figure 3: Specification of class MttsTask

```
DECLARATION:                              INITIAL_STATES:
 state_Mttstask_Obj0      [0,4]           state_MttsTask_Obj0=0
 meth_MttsTask_Obj0       [0,6]           meth_MttsTask_Obj=0
 tkrB_MttsTask_Obj0       [0,1]           tkrB_MttsTask_Obj0=1
 tkwB_MttsTask_Obj0       [0,1]           tkwB_MttsTask_Obj0=1
 pc_MttsTask_Obj0         [0,2]           pc_MttsTask_Obj0=0
 ap_MttsTask_Obj0         [0,5]           ap_MttsTask_Obj0=0
 tkr_MttsTask_Obj0_Ref0   [0,1]           tkr_MttsTask_Obj0_Ref0=0
 tkw_MttsTask_Obj0_Ref0   [0,1]           tkw_MttsTask_Obj0_Ref0=0
```

b) **Transitions.** Each transition has a guard (the enabling condition) and an update expression (executed when the guard holds). Transitions are identified by a unique label.

1. Pulse generates the `evmdd` code below for the constructor of class MttsTask. Prior to the call to the constructor the object does not exist, so the guard below (the part before ->) checks that no permission has been created already for the object. In the update part (the part after ->) all the variables are primed, meaning the after-value of the unprimed variables. The update part sets the program counter to `exe`, which stands for "executing", the method identifier `meth_MttsTask_Obj0` to be a constructor (represented as 1), and grants the object a **Unique** access permission. The update part trans-

fers one token from tkrB_MttsTask_Obj0 to tkr_MttsTask_Obj_Ref0, and from tkwB_MttsTask_Obj0 to tkw_MttsTask_Obj0_Ref0.

```
start_MttsTask_Obj0:
  ap_MttsTask_Obj0=0
    ->
  pc_MttsTask_Obj0'= 1 /* exe */ /\
  meth_MttsTask_Obj0'=1 /* MttsTask */ /\
  ap_MttsTask_Obj0'=1 /* Unique */ /\
  tkrB_MttsTask_Obj0'=0 /\
  tkwB_MttsTask_Obj0'=0 /\
  tkr_MttsTask_Obj0_Ref0'=1 /\
  tkw_MttsTask_Obj0_Ref0'=1
```

2. For method "execute" in Figure 3, Pulse generates the evmdd code below. The guard contains five conjuncts. Therefore, the object must exist, the method cannot be executing, the object typestate should be Ready, some read tokens must be available, and all write tokens must be available. The update expression is composed of seven parts. Therefore, the program counter is set to executing, the method identifier is set to "execute", the access permission is set to **Full**, all the write tokens are transferred, and only one read tokens is transferred.

```
start_execute_MttsTask_Obj0:
 ap_MttsTask_Obj0!=0 /\
 pc_MttsTask_Obj0=2 /* done */ /\
 state_MttsTask_Obj0=1 /* Ready */ /\
 tkrB_MttsTask_Obj0>0 /\
 tkwBMttsTask_Obj0=1
   ->
 pc_MttsTask_Obj0'=2 /* exe */ /\
 meth_MttsTask_Obj0'=3 /* execute */ /\
 ap_MttsTask_Obj0'=2 /* Full */ /\
 tkwB_MttsTask_Obj0=0 /\
 tkrB_MttsTask_Obj0'=tkrB_MttsTask_Obj0-1 /\
 tkr_MttsTask_Obj0_Ref0'=1 /\
 tkw_MttsTask_Obj0_Ref'=1
```

3. The evmdd code below describes the case when method execute transitions from "currently executing" (exe) to "finished" (done), leaving the object in state Complete and setting reading and writing accesses that match the **Full** access permission.

```
end_execute_MttsTask_Obj0:
 pc_MttsTask_Obj0=1 /* exe */ /\
 meth_MttsTask_Obj0=3 /* execute */
   ->
 pc_MttsTask_Obj0'=2 /* done */ /\
```

```
 state_MttsTask_Obj0'= 3 /* Complete */ /\
 tkrB_MttsTask_Obj0'=tkrBMttsTask_Obj0+1 /\
 tkwB_MttsTask_Obj0'=1 /\
 tkr_MttsTask_Obj0_Ref0'=0 /\
 tkw_MttsTask_Obj0_Ref0'=0
```

4. To create an alias, the referenced object must exist, the alias should be afresh, and sufficient read and write tokens should exist to generate a **Pure** access permission (see the guard part). The update expression just creates the reference with a **Pure** access permission.

```
create_alias_MttsTask_Obj0_Ref0:
 state_MttsTask_Obj0!=0 /\
 ap_MttsTask_Obj0=0 /\
 tkrB_MttsTask_Obj0>0
   ->
 pc_MttsTask_Obj0'= 2 /*done*/ /\
 meth_MttsTask_Obj0'=0 /\
 ap_MttsTask_Obj0'= 3 /*Pure*/
```

## 3. Specification and Verification of the MTTS

Our goals for the case study on the specification and verification of the MTTS application using Plural were two-fold: (*i*) to evaluate the design of a massively parallel commercial application like the MTTS, and (*ii*) to determine how well the Plural tool performs on a complex commercial application. The Plural specifications we wrote for the MTTS are based on our understanding of the MTTS application, built from direct inspection of the code of the MTTS, the informal code documentation, and on our discussions with Novabase's engineers. After our discussions with Novabase's engineers took place, we wrote a technical report describing the architecture, the functionality implemented by the MTTS, and the kind of design properties we were interested in checking [13]. The technical report was validated by Novabase and constituted the basis of our work with Plural.

For this case study, we were interested in checking with Plural standard properties related with concurrency such as absence of deadlocks, and the mutual exclusion to a critical section. Checking these properties is essential for the MTTS as it implements programming tasks over threads that are all together accomplishing a common job so that threads need to access shared resources in a synchronized way. It was important too to find out whether Plural could verify these intricate properties or not, or under which circumstances Plural was able to produce a correct answer. We were also interested in checking design properties of the MTTS, properties that must be enforced for the MTTS to be in a validate state, for instance, we were interested in checking whether acquired locks were eventually released. This prompted us to the question on how well Plural deals with reachability properties.

There is an additional value in writing formal specifications for applications. Specifications can be used by engineers to generate a collection of documents describing the behaviour of an application. This documentation can be used by engineers to (re-)

design an application or to resolve differences between members of a quality assurance team regarding the expected behaviour of the application. The process of verifying the MTTS application with Plural revealed a series of issues related to good programming practices and design decisions made in the MTTS that Novabase's engineers used to evolve and improve the MTTS after the case study finished. These issues would not have been revealed otherwise, for example through direct code inspection, which confirms the importance of using automated tools like Plural to increase the confidence on the correctness of the implementation of software applications. For this case study, we kept the code of the MTTS unchanged as much as possible and tried to keep the semantics of the code intact whenever we introduced any changes to it.

### *3.1. Overview of the MTTS Application*

The MTTS is the core of a task distribution server that is used to run tasks over different execution threads. This core is used in the financial sector to process bank checks in parallel with time bound limits. MTTS' implementation is general in the sense that it makes no assumptions on the nature of the running tasks. The MTTS organises tasks through queues and schedules threads to execute the task queues. Tasks are stored in databases. The MTTS is a typical client server application, divided into 3 main components, namely, TaskRegistration, RemoteOperationControl and QueueManager. MTTS' clients use the TaskRegistration component to register tasks. This component stores the registered tasks in a database. The QueueManager component implements some working threads that fetch and execute tasks. The RemoteOperationControl component is used to monitor and to control the progress of the tasks. Every queue implements a mutex manager algorithm to synchronise tasks.

### *3.2. Implementation of the MTTS*

The MTTS is composed of four packages, namely, library, mtts-api, il and server. The library package extends the Java database and Collection classes. The mtts-api package models tasks and queues. The il package models "intelligent locks" and the server package is the main package of the MTTS application and uses features implemented by the other packages as depicted in Figure 4.

Class Task implements tasks and QueueInfo implements queues. Class IMutexImp in the il package implements a mutex algorithm to synchronise tasks, and class MutexManager creates and destroys locks. Lock status and statistics are implemented in classes IMutexStatus and IMutexStatistics respectively. The server package implements code that fetches tasks from the database and distributes them through different threads. Class ServerWrapper runs the server as a system service and Class MttsServer implements the basic functionality to start and stop the server. Class TaskCreator and class QueueManager create tasks and manage queues respectively. Class RemoteTaskRegistration provides an interface to remotely register tasks and class RemoteOperationControl provides an interface to clients to remotely view the progress of tasks. Class ThreadPool keeps a list of class ExecutionThread objects that execute running threads. Class DBConnection implements the basic features to communicate with database.

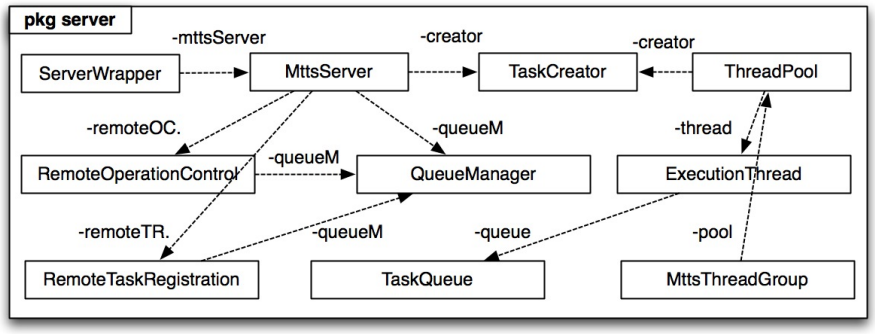


Figure 4: The server package.

### *3.3. The General Specification Approach*

Since Plural performs a modular analysis of programs, we commenced writing specifications starting from the most basic classes of the MTTS, e.g. classes that are inherited from or are used by other classes. Since the specification of more complex classes depends on the specification of the most basic ones, we provided basic classes with a sufficiently detailed specification. We specified the basic packages mtts-api and il first and specified package server last. Because Plural does not include a specification for Java standard classes, e.g. List and Map, we wrote specifications for these Java classes as well. We also wrote specifications for Java classes related to database interaction, e.g. Connection and DriverManager. In the following, we present some of the specifications of the three main packages of the MTTS, discuss various aspects of the specification and verification of the MTTS application with Plural, discuss on the limitations of the Plural tool, and present an evaluation of the case study.

### *3.4. Aspects on the Verification of the MTTS*

We verified various properties of the MTTS application ranging from simple non-null properties to the absence of deadlocks and mutual exclusion to a critical section. Our experience indicates that Plural is a practical tool that can effectively be used to verify complex system properties. In the following, we discuss all these aspects on the specification and verification of properties of the MTTS.

**Processing Tasks.** Figure 2 in Section 2.1.1 presents an excerpt of the specification of class Task. Several aspects of this class's specification are worthy of note. Although we would like to distinguish typestates **Ready** and **Running**, their associated invariants are the same. Plural ensures that if an object is in state **Ready**, then variable data is different than **null**. However, the opposite direction is not necessarily **true**. If one wished to fully distinguish these two typestates then one could add conditions isready and !isready to their respective invariants. But then one would need to modify the source code of class Task by creating a boolean variable isready and to keep track of the value of this variable through the code of class Task. This is error-prone and we

further wanted to keep the source code of the MTTS intact as much as possible. Thus, we left the invariants the same, which is not a problem in Plural.

The specification of class Task ensures that a task cannot be executed twice. Only the class constructor leaves a task in state **Created**. Only method setData transitions a task from **Created** to **Ready**. And a task needs to be in state **Ready** to be executed. The specification also ensures that setData must be called before execute().

**Mutual Exclusion.** Method acquire() acquires a lock and method release() releases the lock. Method acquire() is the only class method that transitions from typestate **NotAcq** into typestate **Acq**, and method release() is the only class method that takes an object from typestate **Acq** into typestate **NotAcq**. Hence, non-nested calls to method acquire() must be preceded by a call to release().

```
@Perm(requires="Full(this) in NotAcq", ensures="Full(this) in Acq")
public abstract void acquire() { }

@Perm(requires="Full(this) in Acq", ensures="Full(this) in NotAcq")
public abstract void release() { }
```

Mutual exclusion to a critical section is thus ensured by enclosing the code of the section between a call to method acquire() and a call to method release() as in method doErrorRecovery() below.

```
@ClassStates({
  @State(name="FullMutex", inv="Full(mutex) in NotAcq"),
})
class ExecutionThread extends Thread {
 private IMutex mutex;

 @Perm(requires = "Full(this) in FullMutex",
       ensures = "Full(this) in FullMutex")
 private void doErrorRecovery(Exception e) {
   try { mutex.acquire(); ... }
   finally { ... mutex.release(); }
 }
}
```

**Absence of Deadlocks.** The Plural specification for methods acquire() and release() ensures that if a thread has acquired a lock, then the thread needs to release the lock before another thread can acquire the lock. This is a source for deadlocks: one needs to check that an acquired lock is eventually released. However, Plural does not provide support for reachability analysis so we were unable to prove the absence of deadlocks in the MTTS implementation in general[4], but only in particular settings, e.g. by direct code inspection. As an example of this, the code of method doErrorRecovery in class ExecutionThread above is enclosed between a call to acquire() and a call to release(). This was often the case for methods in class ExecutionThread. Method doErrorRecovery uses a Java try-catch-finally statement to ensure that the release() method is always finally called regardless of the method termination status (normal or exceptional).

[4]As a consequence of this and other limitations, we developed the Pulse tool [6].

**Destroying a Non-Released Lock.** Method "destroy" in class MutexManager removes a mutex "m" from the list of mutexes. However, destroying (removing) a mutex can lead the system to a deadlock (or to a state that might enable some abnormal behaviour) as the thread that acquired the lock will never be able to release the lock and so other threads waiting for the first thread to release the lock will wait forever. To ensure that a mutex is not destroyed before it is first released, we added the specification "**Full**(m) **in NotAcq**" to the requires part of method destroy. Therefore, Plural will generate an error for any code that calls destroy with a mutex object m in a state other than **NotAcq**.

**Reentrant Mutexes.** According to the implementation and the documentation of the MTTS, although two different threads cannot acquire the same lock, a single thread can acquire the same lock several times. Class IMutexImp implements interface IMutex. It declares a thread field o that keeps track of the thread that owns the lock, and an integer variable nesting that keeps track of the number of times the owner thread has acquired the lock. From the implementation of class IMutexImp, it appears evident that nesting is 0 whenever object "o" is **null** (a class invariant property). We can define two typestates: **NestAcq** (acquired several times by the same thread) and **SingleAcq** (acquired exactly once) as sub-typestates of **Acq**. We can associate **NestAcq** with the invariant "o != **null** * nesting > 1" and use this typestate in the specification of all the methods of the class, e.g. acquire() and release(). However, Plural does not provide support for integer arithmetic. As an alternative, we could modify the code of class IMutexImp to declare and use a boolean variable nested to be **true** whenever nesting is greater than 1, and modify the invariant associated to the typestate **NestAcq** to be "o != **null** * nested==**true**". This approach is potentially error-prone: it requires us to set variable nested accordingly all through class IMutexImp whose code is large.

An additional problem related to the specification of reentrant mutexes has to do with the analysis performed by Plural. We describe this problem with the aid of the specification of the method release() below. This method requires the receiver object to be in state **NestAcq**, so that nested equals to **true**. This indicates that the if-statement in the method release() can never be executed, and hence the receiver object remains in the super-typestate **Acq** (though potentially transitioning from **NestAcq** to **SingleAcq**). However, the Plural tool issues a warning saying that it cannot establish the post-typestate specification. This proves another limitation of the analysis performed by Plural. To determine that the if-statement is never executed, it is necessary to analyse the invariant property associated with the definition of the **NestAcq** typestate. The actual implementation of Plural does not perform such data-flow analysis.

```
// note: only one case (NestAcq) of the specification is shown
@Perm(requires="Full(this) in NestAcq", ensures ="Full(this) in Acq")
public void release() {
  if(o != null && nested==false) { //change to state NotAcq }
  else { //remain in Acq (either NestAcq or SingleAcq) }
}
```

**Plural and Good Programming Practices.** Class ExecutionThread declares a boolean variable "terminate" that is used to determine whether the thread has finished its execution or not. The variable is not explicitly initialized in its declaration, yet according

to the Java specification language its default value is **false**. Typestate **ThreadCreated** represents the state in which a thread has just been created. The constructor of class ExecutionThread does not set variable "terminate". Despite the fact that the initial value of variable "terminate" is **false**, Plural issues an error for the execution of the constructor of class ExecutionThread. This error states that the object cannot be *packed* to typestate **ThreadCreated**.

```
@ClassStates({
 @State(name=``ThreadCreated'', inv=``terminate==false''),
  ...
})
class ExecutionThread extends Thread {
 private boolean terminate;
  ...
 @Perm (ensures = ``Unique(this) in ``ThreadCreated")
 ExecutionThread(...) { ... }
  ...
}
```

Although this shows a bug in the Plural tool, we report it as a programming bad practice. Programmers should explicitly initialize variables to their intended value, thus avoiding relying on the underlying compiler or on external tools, e.g. external typestate analysers like Plural. In this sense, Plural can be used to enforce initialization of class variables. Without proper tool support for analysing large pieces of code, even small bugs like this would remain undetected.

**Specification of Standard Libraries.** The MTTS stores tasks and related information in a database. The DBConnection class of the MTTS implements the basic features that support communication with databases. Plural does not furnish specifications for standard Java classes such as Connection and DriverManager, so we needed to write specifications for these classes as well. The specification of these classes allowed us to prove that the MTTS adheres to general protocols of database interaction. For instance, we proved that a connection is always open whenever database operations such as fetching a task from the database and updating task information stored in the database are taking place.

Abstract class MttsConnection below presents part of the specification we wrote for class MttsConnection. Class MttsConnection defines a root typestate **Connection** with two sub-typestates **OpenConnection** and **ClosedConnection**, modelling an open and closed database connection respectively. According to the Java specification language, method open() can be called on an object to transition it into state **OpenConnection**, and close() can be called on an object to transition it into state **ClosedConnection**. The specification of other standard libraries was conducted in a similar way using Java abstract classes.

```
@Refine({
 @States(dim=``Connection'',
         value={``OpenConnection'', ``ClosedConnection''})
})
public abstract class MttsConnection {
  @Perm(ensures = ``Unique(this) in OpenConnection'')
  MttsConnection() { }

  @Perm(value=``Connection'', ensures=``Full(this) in OpenConnection'')
```

```
  public abstract void open() throws java.sql.SQLException;

  @Perm(value="Connection", ensures="Full(this) in ClosedConnection")
  public abstract void close() throws java.sql.SQLException;
}
```

**Starting and Shutting Down the MTTS Server.** Class MttsServer is the main class of the MTTS application. It implements methods start() and stop() to start and to shutdown the server. It declares three variables OpControlRemote, TaskRegistrationRemote and queueManager to manage the three major features of the server: control of remote operations, task registration, and the queue manager, respectively. One of the design consistency properties of the server is "the server is in the starting state **ServerStart** if and only if its three components are in their starting states **TStart**, **CStart** and **QStart** respectively". And a second consistency property is "the MTTS server is in state **ServerShutdown** if and only its three components are in their shutdown states **TShutdown**, **CShutdown** and **QShutdown** respectively". Although the implementation of method start() verified the definition of typestate **ServerStart**, the implementation of method stop() did not verify the definition of typestate **ServerShutdown**. Therefore, method stop() does not shutdown all its components but only the queueManager.

We report this as a flaw in the design of the MTTS server application. Due to the size of the MttsServer class and all the classes it uses, discovering this design flaw could be difficult with other approaches such as direct code inspection.

```
@ClassStates({
 @State(name="ServerStart",
        inv="Full(queueManager) in QStart *
             Full(OpControlRemote) in CStart *
             Full(TaskRegistrationRemote) in TStart"),
 @State(name="ServerShutdown",
        inv="Full(queueManager) in QShutdown *
             Full(OpControlRemote) in CShutdown *
             Full(TaskRegistrationRemote) in TShutdown")})
class MttsServer {
  private OpControl OpControlRemote;
  private TaskRegistration TaskRegistrationRemote;
  private QueueManager queueManager;

  @Perm(ensures="Full(this) in ServerStart")
  public void start() throws MttsException { ... }

  @Perm(requires="Full(this) in ServerStart",
        ensures="Full(this) in ServerShutdown")
  public void stop() throws MttsException {
   queueManager.shutdown();
  }
  ...
}
```

### *3.5. Discussion on the Limitations of Plural*

In the following, we summarise Plural limitations. Some of these limitations have already been discussed in previous sections.

- The Plural tool approximates the semantics of loops by unrolling them a fixed number of times, which can lead to spurious warnings. We found one case where we had to (unsoundly) approximate a loop with an if statement because Plural ran out of memory when trying to analyze the loop. This is an area where the engineering and scalability of the tool could be improved.

- Plural was not designed to support integer arithmetic. So, one cannot define invariants that use integer variables. Plural provides support for the analysis of boolean expressions that check equality or non-equality of references and boolean expressions that check (non-) nullness of references.

  In the implementation of the MTTS, class IMutexImp implements a mutex algorithm that is used to synchronise threads. Mutexes can be acquired or released. Thus, if a thread acquires a lock then no other thread can acquire the same lock. A thread can acquire a lock several times. So, it must release the lock the same number of times it acquired it for any other thread to (eventually) be able to acquire the lock. However, Plural does not allow one to define a typestate that describes the situation that occurs when a thread has acquired a lock several times as this will require the invariant related to the typestate to rely on an integer arithmetic expression "nesting > 1".

- Plural does not implement a strong specification typechecker, therefore programmers may unintentionally write specifications that include misspelled (nonexistent) typestates, and the Plural analysers can unconsciously use the misspelled typestate in their analysis. Plural does not issue any error on a specification that uses a nonexistent typestate. This feature was intentional—it allows developers to use typestates without declaring them—but nevertheless we found that this feature made it too easy to write incorrect specifications that the tool does not identify.

- Plural does not provide support for reachability analysis, that can be used to answer questions such as if a thread object is in state **Acq**, will the object ever be in state **NotAcq**? In practice, for small classes, one can inspect the code and trace how states evolve manually. For large classes and complex pieces of code this becomes prohibitively difficult.

- Method execute() in Figure 2 transitions a task object from typestate **Ready** to typestate **Running**, and thereafter to typestate **Finished**. These two transitions occur both within method execute(). Typestate **Ready** is required by method execute() and typestate **Finished** is produced by method execute(). **Running** is an intermediate typestate for which the specification of method execute() does not provide any information. It would be possible for execute() to invoke empty helper methods such as startRunning() and stopRunning() that exist merely to specify the two state transitions, however this is at best an awkward workaround. Not being able to reason more cleanly about *intermediate* program states is a limitation of the analysis performed by the Plural tool. The information about intermediate states could be used by programmers (and tools) to assert certain facts that otherwise cannot be asserted explicitly. For instance, in the verification

of the MTTS, we could not specify the property stating that a running task cannot be *deleted*.

### *3.6. Evaluation of the Case Study*

It has taken approximately nine months to undertake this case study. The three first months were spent on developing a precise understanding of the MTTS. During this process, we communicated by email with the Novabase engineers and conducted face-to-face meetings. The other six months were spent on writing the specifications for the MTTS. Most of the specifications are written by the second author, who beforehand did not have any experience with Plural or the Plural language, or with writing formal specifications in general. The first author, who had experience in writing formal specifications with JML [14, 15], but not with Plural, supervised the work and made suggestions to improve the specifications. The third author, the principal designer of Plural, provided feed-back on writing Plural specifications.

We specified and verified forty nine Java classes with 14451 lines of Java code and 546 lines of program specifications. The automation of the analyses performed by the Plural tool ranges from a couple of milliseconds for the verification of small classes to a couple of minutes for the verification of large classes.

## 4. Model Checking Plural Specifications

### *4.1. Methodology*

The Plural tool takes as input a specification, an implementation of that specification, and optionally a client program. The implementation and client are checked against the specification. An issue stemming from this style of verification is that if the specification itself has errors or unintended semantics, the programmer might never become aware of it. For example, if the specification is tested against just one client, the specification may preclude behavior that was intended to be possible but did not show up in that client, and the tool will not point this out.

We have developed an approach to identifying issues with Plural specifications. In our approach, the behavior of object oriented concurrent programs is defined as a maximally unconstrained interleaving of threads. The interleavings are represented as sequences of method calls obeying access permission rules and typestate definitions.

We use the high-performance symbolic model checker `evmdd-smc`[5] [9], developed for NASA by the National Institute of Aerospace. The input language of `evmdd-smc` is similar to SAL (Symbolic Analysis Laboratory), however `evmdd-smc` is more efficient than SAL for several reasons. The `evmdd-smc` is powered by an edged-valued decision diagrams (EVMDD) library, `libevmdd`[5], which can be orders of magnitude faster than the ubiquitous CUDD, especially for models that capture concurrency [9]. Secondly, the much leaner `evmdd-smc` is free of all of the syntactic sugar provided by SAL, which often leads to tremendous pre-processing overhead.

[5] Available at `http://research.nianet.org/~radu/evmdd`.

*4.2. Abstract Models of Plural Specifications.*

Our abstract state-machine models attempt to capture the collective dynamic behaviour of the object references described in Plural specifications in a concurrent environment. The models are not concerned with the body of methods, but only with their specifications – i.e., preconditions, postconditions, typestates, and access permissions – and the interleavings of concurrently executing processes.

A Plural specification comprises a finite set of class declarations $C = \{C_1, \ldots, C_c\}$. Every class $C_i$ contains a set of typestate declarations, $\mathcal{TS}_i = \{t_i^1, \ldots, t_i^{l_i}\}$, where $l_i$ is the number of typestates of class $C_i$, for $1 \leq i \leq c$. We identify each object that appears in the specification (including method parameters and class fields) $O = \{o_1, \ldots, o_n\}$. Each object is mapped to its class declaration via an implicit mapping $class\_of : O \rightarrow C$. To capture the issue of concurrency, for each object $o_i$, $1 \leq i \leq n$, we create a number of instances of references to that object: $\mathcal{R}_i = \{r_i^0, r_i^1, \ldots, r_i^j, \ldots, r_i^K\}$. $K$ is a parameter that can be set by the user to a desired value. For $K = 0$, there is no concurrency in the model, while for any strictly positive value, $K$ other independent aliases will be introduced for each reference, corresponding to a truly concurrent setting.

The question whether there exists a smallest value for $K$ that is sufficiently large to capture all "relevant" behaviors depends on the expressiveness of the specification logic. If integer arithmetic is allowed for typestate definitions (for invariant definitions), it is easy to construct a model where no such smallest upper bound exists, for example by defining a typestate that is entered when the reference count to the object exceeds a certain value $n$. In this implementation, we do not include integer arithmetic in the invariant expressions. However, the existence of an upper bound for $K$ in this case, while possible, has not been determined yet.

*The Basic Component.* In the following, we reserve the symbol $\perp$ for undefined values (for multiple domains: typestates, access permissions, methods), throughout the paper.

The building block of the developed model is the state-machine of an object reference $r_i^j$, where $h = class\_of(i)$, which includes:

(a) the abstract program counter, $(pc_i^j) \in \mathcal{PC}_i = \{exe, done\} \times (\{\perp\} \cup \mathcal{M}_h)$, $\mathcal{M}_h = \{M_h^1, \ldots M_h^{m_h}\}$, is the set of methods defined for object $o_i$, including constructors.

(b) the access permissions associated with $r_i^j$: a field of enumerated type $ap_i^j \in \mathcal{AP} = \{\perp, \textbf{Unique}, \textbf{Full}, \textbf{Pure}, \textbf{Immutable}, \textbf{Share}\}$.

(c) additionally, each object has an associated typestate $ts_i \in \mathcal{TS}_h = \{\perp\} \cup \{t_h^1, \ldots, t_h^{l_h}\}$,

*State Transition Rules.* The model allows a non-deterministic transition from *done*-local-states (i.e. a local states with $pc_i^j = (done, \cdot)$) to any other *exe*-local-state provided it respects its transition guards. This covers all possible sequences of method calls for $K + 1$ concurrently executing references, which is behaviorally equivalent to placing the reference $(this, o_i)$ in any reachable global context with $K$ other references. The transitions from *done*-local-states to *exe*-local-states are guarded by expressions that capture:

(i) the required typestate condition of the *exe*-local-state

(ii) the access permission constraints determined by the splitting rules described in subsection 5.1.

Additionally, from each *exe*-local-state $(exe, m)$ a reference can only transition to its matching *done*-local-state $(done, m)$, capturing the completion of the call to method $m$. The transition is guarded by the postcondition associated with the method in the specification and reflects the change in typestate and access permissions that may occur.

## 5. The Translation Algorithm

The translation algorithm from Plural to `evmdd` builds the two components of a finite state machine: the set of potential global states $S$ and the transition relation between states, $R \subseteq S \times S$. The potential state space is simply the cross product of the local state spaces of all $n$ objects. This includes the typestate of the object (common to all references to that object), and the program counter and access permission for each of the $K + 1$ references:

$$S = \prod_{i=1}^{n} \left( \mathcal{TS}_i \times \prod_{j=0}^{K} (\mathcal{PC}_i \times \mathcal{AP}) \right)$$

The transition relation can be defined component-wise. For each reference $r_i^j$ there are two local transitions, corresponding to starting a method and ending a method. The global transition relation is the asynchronous composition of the local transition relations. We use the standard notation for pairs of states (*from*-states, *to*-state) in the transition relation, where unprimed variables refer to the *from*-state and primed variables to the *to*-state. We also employ the following type definitions:

| | | |
|---|---|---|
| *GlobalTypestate* | = | $TS_1 \times \ldots \times TS_n$ |
| *LocalState* | = | $(PC, AccessPermission)$ |
| *GlobalState* | = | Array[1..$n$] of Array [0..$K$] of *LocalState* |
| *Reference* | = | $(ObjectIdx, AliasIdx)$ |
| *Triple* | = | $(Reference, Typestate, AccessPermission)$ |

The routines *StartMethod* and *EndMethod* listed in Algorithms 1 and 2 are used to construct the transition relation. They correspond to starting and ending a method $m$ by a reference $r_i^j$, respectively. The input for these routines are the calling reference $r_i^j$, the method $m$, the global context (the global state $s$ and the typestate $t$), and two triples. The triples $\left(r_{i_0}^{j_0}, ts_{i_0}^{k_0}, ap_0\right)$ and $\left(r_{i_1}^{j_1}, ts_{i_1}^{k_1}, ap_1\right)$ encode the **requires** (indexed $i_0$) and **ensures** (indexed $i_1$) clauses from the method's specification, i.e. the required and ensured typestate, reference, and access permission. The output of the routines are two Boolean formulae: *guard* and *update*. The *guard* formula must hold for the transition to be enabled, and the *update* formula encodes the changes in the values of global states that occur by executing a transition.

For *StartMethod*, if the reference $r_i^j$ exists, and it is not already executing a method, and the global type state of object $i_0$ is the required typestate $ts_{i_0}^{j_0}$, and the required access permission of object $i_0$ is compatible with the current $ap_0$, and the ensured access permission of object $i_1$ is compatible with $ap_1$ (all conjuncts are part of the

**Algorithm 1** for the transition corresponding to starting a method

StartMethod( $s : \mathit{GlobalState}, t : \mathit{GlobalTypestate}, r_i^j : \mathit{Reference}, m : \mathit{Method}_i,$
$\left(\left(r_{i_0}^{j_0}, ts_{i_0}^{k_0}, ap_0\right), \left(r_{i_1}^{j_1}, ts_{i_1}^{k_1}, ap_1\right)\right) : \mathit{Triple} \times \mathit{Triple}$
)

$$\begin{aligned} \mathit{guard} \quad &\leftarrow \quad s[i][j].ap \neq \bot \wedge s[i][j].pc = (\mathsf{done}, \cdot) \wedge t[i_0] = ts_{i_0}^{j_0} \wedge \\ &\quad\; \mathit{Compatible}\,(s[i_0][j_0].ap, ap_0) \wedge \mathit{Compatible}\,(s[i_1][j_1].ap, ap_1) \\ \mathit{update} \quad &\leftarrow \quad s'[i][j].pc = (\mathsf{exe}, m) \wedge \mathit{ChangePermission}\,(s[i_0][j_0], ap_0) \end{aligned}$$

**return** $\mathit{guard} \Rightarrow \mathit{update}$

**Algorithm 2** for the transition corresponding to ending a method.

EndMethod( $s : \mathit{GlobalState}, t : \mathit{GlobalTypestate}, r_i^j : \mathit{Reference}, m : \mathit{Method}_i,$
$\left(\left(r_{i_0}^{j_0}, ts_{i_0}^{k_0}, ap_0\right), \left(r_{i_1}^{j_1}, ts_{i_1}^{k_1}, ap_1\right)\right) : \mathit{Triple} \times \mathit{Triple}$
)

$$\begin{aligned} \mathit{guard} \quad &\leftarrow \quad s[i][j].pc = (\mathsf{exe}, m) \\ \mathit{update} \quad &\leftarrow \quad t'[i_1] = ts_{i_1}^{k_1} \wedge s'[i_1][j_1].ap = ap_1 \wedge s'[i][j].pc = (\mathsf{done}, m) \wedge \\ &\quad\; \mathit{ChangePermission}\,(s[i_1][j_1].ap, ap_1) \end{aligned}$$

**return** $\mathit{guard} \Rightarrow \mathit{update}$

guard), then we update the status of the program counter of $r_i^j$ to be "executing method $m$", and we update the global state.

For *EndMethod*, if method $m$ is the one currently executed (the guard), then we update the typestate of $r_{i_1}^{j_1}$ to its ensured typestate, its access permission to $ap_1$, and the status of the program counter of the calling reference $r_i^j$ to "done",

In the special case when $m$ is a constructor, the guard for *StartMethod* is slightly different: the first predicate, $s[i][j].ap \neq \bot$, enforcing that the reference $r_i^j$ exists, is replaced by $t[i] = \bot$, that enforces the opposite: the object $o_i$ has not been created yet.

There are two routines that require further explanations. $\mathit{Compatible}(ap_x, ap_y)$, implements a Boolean function that decides whether the access permissions $ap_x$ and $ap_y$ are "compatible", i.e. if $ap_x$ can be downgraded or upgraded to $ap_y$ according to Section 5.1. The second routine, $\mathit{ChangePermission}(ap_x, ap_y)$, builds the update formula corresponding to a compatible access permission transformation from $ap_x$ to $ap_y$.

### *5.1. Access Permission Compatibility and Transformation Rules*

Object permissions must guarantee that an object reference having *write* permission does not conflict with another reference to the same object having *write* or a *read* permission, e.g race conditions should not occur. The presence of object aliases makes the analysis of permissions harder. To enhance this analysis, permissions in Plural cannot be duplicated and can be *split* in fractions of a permission. A permission can take a fractional value in the interval $(0, 1]$. Fractional permission analysis as implemented in Plural has been influenced by Boyland's work in [2]. Representing permissions as fractions explains when writes permissions conflict with other permissions. A write permission requires to have the whole fraction of the object permission to write to the

$$
\begin{aligned}
\textbf{Unique}(x,o,k) &\iff \textbf{Full}(x_1,o,k_1) \otimes \textbf{Pure}(x_2,o,k_2)\\
\textbf{Unique}(x,o,k) &\iff \textbf{Share}(x_1,o,k_1) \otimes \textbf{Share}(x_2,o,k_2)\\
\textbf{Full}(x,o,k) &\iff \textbf{Immutable}(x_1,o,k_1) \otimes \textbf{Immutable}(x_2,o,k_2)\\
\textbf{Immutable}(x,o,k) &\iff \textbf{Pure}(x_1,o,k_1) \otimes \textbf{Immutable}(x_2,o,k_2)\\
\textbf{Immutable}(x,o,k) &\iff \textbf{Immutable}(x_1,o,k_1) \otimes \textbf{Immutable}(x_2,o,k_2)
\end{aligned}
$$

Figure 5: Access permission splitting rules

object, so two aliases of an object cannot simultaneously write to an object or read and write to it. On the other hand, two read permissions can coexist. Our implementation of fractions as described in Section 5.2 uses the bounding assumption of a $K$ maximum number of object references, and thus permissions take values of a fraction in the range $(0, K]$. In Plural, a **Unique** permission is represented by 1 and an **Immutable** permission is represented by a fraction between 0 and 1 so that other permissions are allowed to exist. Figure 5 presents all the splitting rules Plural enforces. For all of the listed rules, $k = k_1 + k_2$ and at least one of the references $x_1$ and $x_2$ is $x$.

| Next AP | **Unique** | **Full** | **Share** | **Immutable** | **Pure** | $\perp$ |
|---|---|---|---|---|---|---|
| Current AP | **this O**<br>rw rw | **this O**<br>rw rw | **this O**<br>rw rw | **this O**<br>rw rw | **this O**<br>rw rw | **this O**<br>rw rw |
| **Unique** | $\leftrightarrow$<br>== == | $\downarrow$<br>== + = | $\downarrow$<br>== ++ | $\downarrow$<br>= − + = | $\downarrow$<br>= − ++ | $\downarrow$<br>−− ++ |
| **Full** | $\uparrow$<br>== − = | $\leftrightarrow$<br>== == | $\downarrow$<br>== = + | $\downarrow$<br>= − == | $\downarrow$<br>= − = + | $\downarrow$<br>−− = + |
| **Share** | $\uparrow$<br>== −− | $\uparrow$<br>== = − | $\leftrightarrow$<br>== == | $\otimes$<br>= − = − | $\downarrow$<br>= − == | $\downarrow$<br>−− == |
| **Immutable** | $\uparrow$<br>= + − = | $\uparrow$<br>= + == | $\otimes$<br>= + = + | $\leftrightarrow$<br>== == | $\downarrow$<br>== = + | $\downarrow$<br>− = = + |
| **Pure** | $\uparrow$<br>= + −− | $\uparrow$<br>= + = − | $\uparrow$<br>= + == | $\uparrow$<br>== = − | $\leftrightarrow$<br>== == | $\downarrow$<br>− = == |
| $\perp$ | $\uparrow$<br>++ −− | $\uparrow$<br>++ = − | $\uparrow$<br>++ == | $\uparrow$<br>+ = = − | $\uparrow$<br>+ = == | $\leftrightarrow$<br>== == |

Figure 6: Access Permission Transformations (downgrade/upgrade).

Figure 6 shows the routines *Compatible*() and *ChangePermission*() presented in the beginning of Section 5. There can be two generic ways of access permission transformations. *i*. downgrade: this reference may give up rights and the other references may gain rights. *ii*. upgrade: this reference may acquire more rights and the other references may lose rights. The directional arrows denote the nature of the transformations: $\downarrow$ for downgrade, $\uparrow$ for upgrade, $\leftrightarrow$ for no change, and $\times$ for disallowed. The second row of symbols describes the read and write permissions change for reference **this** and the other references ("the rest of the world" denoted by **O**): + for gaining, − for losing, = for no change. Therefore, a **Full** permission is compatible with a **Share** permission.

Additionally, **Full** can be downgraded to **Share** if **this** gives up writing permissions that are gained by the other references. Our analysis has found a single case of incompatibility, between **Share** and **Immutable** permissions, so they cannot be transformed one from another.

*5.2. A Discrete State Semantics for Fractional Access Permissions*

The access permission transformations are based on an underlying concept of "collective management" of permissions among references to the same object. Intuitively, access permissions are viewed as resources (tokens), stored in a central location (bank) and available globally. The references can take a portion (fraction) or all tokens, depending on access's needs and then return them back to the bank.

We describe the access permissions of a reference $r_i^j$ as a pair of fractions: $(fr_i^j, fw_i^j)$, with $fr_i^j, fw_i^j \in [0, 1]$, representing the fraction of the read and write permissions to object $o_i$ owned by reference $r_i^j$. There are three semantic classes for the values of a fraction $f$: $f = 0$ (no permission), $0 < f < 1$ (partial/shared permission), or $f = 1$ (exclusive rights). The preservation of access permissions is a global invariant, with $fr_i^B$ and $fw_i^B$ the unused fractions (still in the bank): $fr_i^B + \sum_{j=0}^{K} fr_i^j = 1 \wedge fw_i^B + \sum_{j=0}^{K} fw_i^j = 1$

| **this** | **Semantic** | **Bank** | **O** |
|---|---|---|---|
| $fr_{i_i}^j = 0 \wedge fw_{i_i}^j = 0$ | **null** | $fr_i^B \geq 0 \wedge fw_i^B \geq 0$ | any |
| $fr_i^j = 0 \wedge fw_i^j > 0$ | no meaning | - | - |
| $0 < fr_{i_i}^j < 1 \wedge fw_{i_i}^j = 0$ | **Immutable** | $fw_i^B = 1$ | $\sum_{l \neq j} fw_i^l = 0$ |
| $0 < fr_{i_i}^j < 1 \wedge fw_i^j = 0$ | **Pure** | $fw_i^B < 1$ | $\forall l \neq j : fw_i^l \geq 0$ |
| $0 < fr_i^j, fw_i^j < 1$ | **Share** | $0 \leq fr_i^B < 1 \wedge 0 \leq fw_i^B < 1$ | $\forall l \neq j : 0 \leq fr_i^l, fw_i^l < 1$ |
| $0 < fr_i^j < 1 \wedge fw_i^j = 1$ | **Full** | $0 \leq fr_i^B < 1 \wedge fw_i^B = 0$ | $\forall l \neq j : fw_i^l = 0$ |
| $fr_{i_i}^j = 1 \wedge fw_{i_i}^j = 0$ | **Immutable** | $fr_i^B = 0 \wedge fw_i^B = 1$ | $\forall l \neq j : fr_i^l = 0 \wedge fw_i^l = 0$ |
| $fr_{i_i}^j = 1 \wedge fw_i^j = 0$ | **Immutable** | $fr_i^B = 0 \wedge fw_i^B < 1$ | $\forall l \neq j : fr_i^l = 0 \wedge fw_i^l > 0$ |
| $fr_{i_i}^j = 1 \wedge 0 < fw_i^j < 1$ | no meaning | - | - |
| $fr_i^j = 1 \wedge fw_i^j = 1$ | **Unique** | $fr_i^B = 0 \wedge fw_i^B = 0$ | $\forall l \neq j : fr_i^l = 0 \wedge fw_i^l = 0$ |

Figure 7: A fractional permission model

The possible combinations of values for fractions are listed in Figure 7. The meaningless combinations arise from the implicit subordination of read permissions to modifying (write) permissions: a reference with modifying permissions has to have reading permissions as well. Also note that the nature of others rights can be inferred from the value of **this** and the bank: $\sum_{l \neq j} fr_i^l = 1 - (fr_i^j + fr_i^B)$ and $\sum_{l \neq j} fw_i^l = 1 - (fw_i^j + fw_i^B)$. This helps determine locally the evaluation of the quantified formulae in the definition of certain access permissions without having to consult the actual values of the other references. This has practical importance for model checking in particular, where *event locality* can impact the efficiency of the analysis.

Our implementation uses the bounding assumption of maximum $K$ co-existing references to translate this framework into a fully discrete model, where we map fractions from the continuous interval $[0, 1]$ to the set $\{0, 1, \ldots, K + 1\}$, via the abstraction

$$N : [0, 1] \to \{0, 1, \ldots, K{+}1\}, \quad N(f) = \begin{cases} 0, & \text{if} \quad f = 0 \\ x \in \{1, \ldots, K\}, & \text{if} \quad 0 < f < 1 \\ K{+}1, & \text{if} \quad f = 1 \end{cases}$$

We can define two functions for the required number of tokens needed for the next operation, $N_r$ and $N_w$. There are multiple ways to define this pair of functions, as there is still non-determinism in the abstraction from fractions to integer values. The definition below corresponds to the most conservative approach in which references request the minimum amount of resources required for their operation:

$$N_r : \mathcal{AP} \to \{0, .., K{+}1\}, N_r(a) = \begin{cases} 0, & \text{if} \quad a = \bot \\ 1, & \text{if} \quad a \in \{\textbf{Full}, \textbf{Pure}, \textbf{Immutable}, \textbf{Share}\} \\ K{+}1, & \text{if} \quad a = \textbf{Unique} \end{cases}$$

$$N_w : \mathcal{AP} \to \{0, \ldots, K{+}1\}, N_w(a) = \begin{cases} 0, & \text{if} \quad a \in \{\bot, \textbf{Pure}, \textbf{Immutable}\} \\ 1, & \text{if} \quad a = \textbf{Share} \\ K{+}1, & \text{if} \quad a \in \{\textbf{Unique}, \textbf{Full}\} \end{cases}$$

To complete our model, in addition to the field $ap$ in the basic module, we introduce $tkr$ and $tkw$, to represent the number of read and write tokens for each reference $q = r_i^j$.

For example, if a method requires non-exclusive rights (**Pure**), the guard for starting the method checks whether the reference has the one read token necessary (a) or it needs to "borrow" it (b). This results in two distinct types of transitions:

(a) If $tkr_i^B + tkr_i^j \geq 1$, then $tkr'^{\,j}_{\,i} = 1 \wedge tkr'^{\,B}_{\,i} = tkr_i^B + tkr_i^j - 1$

(b) If $tkr_i^B + tkr_i^j = 0 \wedge \exists h \neq j : pc_i^h = (\text{done}, \cdot) \wedge tkr_i^h \geq 1$, then $tkr'^{\,j}_{\,i} = 1 \wedge tkr'^{\,h}_{\,i} = tkr_i^h - 1$ (one read token transferred from $r_i^h$ to $r_i^j$)

*5.3. The Model Generator.*

An `evmdd` model contains four sections: variable declarations, variable initialisations, the transition relation, and a set of CTL temporal logic [16] properties.

a) *Abstract variable domains.*

As all variables in the model have to be discrete (more precisely of an integer interval type) we have to define the abstract domains for all of them. The domains $\mathcal{TS}_i = \{\bot\} \cup \{t_i^1, \ldots, t_i^{h_i}\}$ are mapped to $[0, h_i]$. The domain of method identifiers $\{\bot\} \cup \{M_i^1, \ldots, M_i^{m_i}\}$ is mapped to $[0, m_i]$. The domain of access permission types $\mathcal{AP} = \{\bot, \textbf{Unique}, \textbf{Full}, \textbf{Pure}, \textbf{Immutable}, \textbf{Share}\}$ is mapped to $[0, 5]$. For the variables referring to tokens, we use the domain $[0, K + 1]$. For the $\{exe, done\}$ type we use $[0, 1]$.

b) *Variable declarations.*

For each object $o_i$, we declare two categories of variables. One category refers to the proper object and includes three variables: $state_i$ of type $\mathcal{TS}_i$, $tkr_i^B$ and $tkw_i^B$ of type $[0, K + 1]$. The second category is for references to the object. For each of the $K + 1$ references to $o_i$, we define five variables: $pc_i^j$ of type $[0, 1]$, $method_i^j$ of type $[0, m_i]$, $ap_i^j$ of type $[0, 5]$, $tkr_i^j$ and $tkw_i^j$ of type $[0, K + 1]$. Hence, we have $c * (3 + 5 * (K + 1))$ variables in the model.

c) *Variable initializations.*

For each object: $state_i = \perp(0) \wedge tkr_i^B = K + 1 \wedge tkw_i^B = K + 1$.

For each reference: $0 \leq j \leq K$: $pc_i^j = \mathsf{done}(1)$, $method_i^j = \perp(0)$, $ap_i^j = \perp(0)$, $tkr_i^j = tkw_i^j = 0$.

d) *Transition relation.*

Each transition has a guard expression (the enabling condition) and an update expression (the transformation performed by executing the transition). Recall that unprimed variables refer to values before the update (the "from" state), while primed variables refer to values after the update (the "to" state). We identify four categories of transitions in our model, described in detail below.

1. Reference $r_i^j$ starts a constructor.
   The guard enforces that the object has not been created: $\bigwedge_{j=0}^{K}(ap_i^j = \perp)$.
   The update expression sets the program counter and method of $r_i^j$ and takes all $K + 1$ tokens of both types (read and write) from the bank: $pc'_i{}^j = \mathsf{exe} \wedge method'_i{}^j = \mathsf{constructor} \wedge ap'_i{}^j = \textbf{Unique} \wedge tkr'_i{}^B = 0 \wedge tkw'_i{}^B = 0 \wedge tkr'_i{}^j = K + 1 \wedge tkw'_i{}^j = K + 1$.
2. Reference $r_i^j$ starts a method $m_i^k$.
   The guard contains four conjuncts. The first requires $r_i^j$ to exist (i.e., not undefined): $ap_i^j \neq \perp$. The second requires it to be in a $\mathsf{done}$ state (i.e. not executing something else): $pc_i^j = \mathsf{done}$. The third conjunct enforces that: $state_h = t_h^x$ and the fourth conjunct checks the availability of access permission tokens. As explained in the definition of $N_r$ and $N_w$, there may be 0, 1, or $K + 1$ tokens requested for read and/or write permissions associated with method $m_i^k$. In general, if $tr_i^k$ and $tw_i^k$ are the number of tokens needed to execute method $m_i^k$, then the fourth conjunct is: $tkr_i^B \geq tr_i^k \wedge tkw_i^B \geq tw_i^k$. Note that if $tr_i^k = 0$, the expression $tkr_i^B \geq tr_i^k$ is always true, hence it can be ignored. The same observation holds for the case $tw_i^k = 0$. The update expression has two conjuncts. The first reflects the changes in the state of $r_i^j$: $pc'_i{}^j = \mathsf{exe} \wedge method'_i{}^j = k \wedge ap'_i{}^j = ap$. The second reflects the changes in the distribution of tokens: $tkr'_i{}^B = tkr_i^B - tr_i^k \wedge tkw'_i{}^B = tkw_i^B - tw_i^k \wedge tkr'_i{}^j = tkr_i^j + tr_i^k \wedge tkw'_i{}^j = tkw_i^B + tw_i^k$.
3. Reference $r_i^j$ ends a method $m_i^k$.
   The guard ensures that the reference is actually executing method $m_i^k$: $pc_i^j = \mathsf{exe} \wedge method_i^j = k$. The update expression reflects the change in the state of $r_i^j$: $pc'_i{}^j = \mathsf{done}$, and returns all the access permission tokens held by $r_i^j$ back to the bank: $tkr'_i{}^B = tkr_i^B + tr_i^k \wedge tkw'_i{}^B = tkw_i^B + tw_i^k \wedge tkr'_i{}^j = tkr_i^j - tr_i^k \wedge tkw'_i{}^j = tkw_i^B - tw_i^k$. If the specification of $m_i^k$ also ensures that some object $o_h$ (again, not necessarily the same $o_i$) is left in state $t_h^x$, the second conjunct enforces that: $state'_h = t_h^x$.

4. Reference $r_i^j$ is a newly created alias.
   The guard expression requires that the object exists, $state_i \neq \bot$, $r_i^j$ has not been previously created, $ap_i^j = \bot$, and enough read tokens exist for a pure access, $tkr_i^B \geq 1$, which is the most conservative approach.
   The update expression is ${pc'}_i^j = \mathsf{done} \wedge {method'}_i^j = \bot \wedge {ap'}_i^j = \textbf{Pure}$.

## 6. Running Pulse on the Specification of the MTTS

### *6.1. Checking the Absence of Sink States (global deadlocks)*

The presence of states without successors (sink states) may have different root causes, including improper use of access permissions that block the progress of all threads, among which deadlock (due to a mutual circular wait) is one particular undesired behaviour. In the CTL temporal logic, this can be expressed as the property *deadlock* : ¬EX(*true*). Note that local deadlocks may exist even when there are no global sinks, but testing for all local deadlocks requires an exponential number of tests.

### *6.2. Checking the Absence of Unreachable Methods*

Pulse uses the information of the **requires** clause of a method's specification to check whether it can be reached from another method or not. In our model, we represent with the predicate $satisfiability_i(m_n)$ to check whether the precondition of the method $m_n$ is satisfiable.

$$\forall 1 \leq i \leq c, \forall m_n \in M_i\text{: } satisfiability_i(m_n) : \mathrm{EX}(pc_i^j = (m_n, exe))$$

When we run Pulse on the specifications of Figure 3, Pulse states that three methods setData, execute, and delete are unreachable because of the unsatisfiability of the **requires** clauses of these methods. The **ensures** clause of MttsTask constructor produces an object with **Unique** access permission and in typestate Created. A **Unique** access permission can be *transformed* into two access permissions i.e **Full** and **Pure** (according to the fractional permission rules). The **Full** access permission and the typestate Created can be used to match the first part of the **requires** clause of method setData, however the second part of the **requires** clause i.e the parameter "d" should be in state Filled can not be matched, as none of the methods defined in class MttsTaskDataX transitions into typestate Filled, and hence as a result the method setData is unreachable. The unsatisfiability of the setData **requires** clause will not produce the **ensures** typestate i.e typestate Ready. The non-availability of the typestate Ready will lead to the unsatisfiability of the **requires** clause of the method execute, so the method execute will remain unreachable. Similarly the unreachability of the method execute will result into the non-availability of the the typestate Complete that will lead to the unreachability of the method delete.

### *6.3. Checking Whether Methods Can be Executed in Parallel*

Access permissions can be used to represent parallel executions of methods $m_1$ and $m_2$ along with other dependency information. Pulse can find all possible pairs of methods that can be executed in parallel and all pairs of methods that can never be

| | MttsTask | setData | getData | execute | delete | getTaskStatus |
|---|---|---|---|---|---|---|
| MttsTask | ⋕ | ⋕ | ⋕ | ⋕ | ⋕ | ⋕ |
| setData | ⋕ | ⋕ | ⋕ | ⋕ | ⋕ | ⋕ |
| getData | ⋕ | ⋕ | ⋕ | ⋕ | ⋕ | ‖ |
| execute | ⋕ | ⋕ | ⋕ | ⋕ | ⋕ | ⋕ |
| delete | ⋕ | ⋕ | ⋕ | ⋕ | ⋕ | ⋕ |
| getTaskStatus | ⋕ | ⋕ | ‖ | ⋕ | ⋕ | ⋕ |

Table 1: Method Concurrency Matrix

| **Packages** | **Classes** | **Methods** | **State Space** | **#Properties** | | |
|---|---|---|---|---|---|---|
| | | | | SS | MR | STM |
| `library` | 8 | 39 | $1 \times 10^8$ | 1 | 39 | 6 |
| `il` | 13 | 61 | $7 \times 10^9$ | 1 | 61 | 96 |
| `mtts` | 19 | 166 | $2 \times 10^{16}$ | 1 | 166 | 98 |
| `il`, `library` | 21 | 100 | $1 \times 10^{18}$ | 1 | 100 | 102 |
| `il`, `library`, `mtts` | 40 | 266 | $2 \times 10^{34}$ | 1 | 266 | 200 |
| `il`, `library`, `mtts`, `server` | 55 | 368 | $8 \times 10^{52}$ | 1 | 368 | 280 |

Table 2: Pulse Results showing State Space and Checked Properties of the MTTS specification.

executed in parallel. In CTL this can be expressed as $\forall 1 \leq i \leq c, 0 \leq j_1 \neq j_2 \leq K, \forall m_1 \neq m_2 \in \mathcal{M}_i$:

$$concurrent_i\,(m_1, m_2) : \mathrm{EF}\left(pc_i^{j_1} = (m_1, exe) \wedge pc_i^{j_2} = (m_2, exe)\right)$$

An empty set of states satisfying $concurrent_i\,(m_1, m_2)$ indicates that $m_1$ and $m_2$ can never be executed in parallel.

When we run Pulse on the specifications of Figure 3, Pulse produced the concurrency matrix as shown in the Table 1. The symbol ‖ indicates that the methods can be executed parallel and the symbol ⋕ indicates that methods cannot be executed in parallel. For instance, no methods can be executed in parallel with the constructor MttsTask; similarly, methods that requires **Full** (modifying) access permissions, e.g. execute and delete, cannot be executed in parallel. However, the methods that requires **Pure** (read-only) access permissions, e.g. getData and getTaskStatus, can be executed in parallel with each other.

### *6.4. Performance Analysis*

Tables 2 and 3 show the results produced by Pulse on the full MTTS specification. In both tables, SS stands for sink states, MR for method reachability and STM

| Packages | Classes | Methods | Runtime(s) | | | Violations | | |
|---|---|---|---|---|---|---|---|---|
| | | | SS | MR | STM | SS | MR | STM |
| `library` | 8 | 39 | 0.07 | 0.30 | 0.04 | 0 | 0 | 1 |
| `il` | 13 | 61 | 0.10 | 0.18 | 0.09 | 0 | 0 | 9 |
| `mtts` | 19 | 166 | 0.11 | 0.33 | 0.17 | 0 | 44 | 26 |
| `il`, `library` | 21 | 100 | 0.08 | 0.25 | 0.15 | 0 | 0 | 27 |
| `il`, `library`, `mtts` | 40 | 266 | 0.43 | 0.89 | 0.44 | 0 | 44 | 37 |
| `il`, `library`, `mtts`, `server` | 55 | 368 | 24.34 | 152.39 | 2824.47 | 0 | 58 | 60 |

Table 3: Pulse Results showing Time and Detected Violations of the MTTS specification.

for state transition matrix. Table 2 provides metrics about state space and number of checked properties. For an instance, Pulse used a state space of $8 \times 10^{52}$ to check all the properties of the whole MTTS specification. In Table 3, columns 4 to 6 show the time taken by Pulse (in seconds) to perform the tests. The first five rows represent partial models when the three utility packages are checked in isolation. The last row includes the `server` package that has most of the class interdependencies and hence there is a significant jump in the time (Runtime), taken by Pulse to verify the properties. The last three columns show the number of violations found, more precisely, the number of sink sates, unreachable methods, and unreachable typestates, respectively. For an instance, Pulse found 58 unreachable methods and 60 unreachable typestates in whole MTTS specifications. The main reasons of these violations are (*a*) the writing of wrong specifications that lead to unreachable typestates or methods. We also observe that sometimes, the specifications of complex code (that involves many typestates) lead to wrong (unwanted) transitions, (*b*) the specification of class constructors that do not produce access permissions. Wrongly specified constructors lead to a situation in which class method preconditions are never satisfied, and (*c*) added specification on a method's parameter but then necessary definition of parameter's typestate is missing, in the respective parameter class.

## 7. Related Work

In previous work [17], we used JML to specify an electronic purse application written in the Java Card dialect of Java. JML is a behavioural interface specification language for Java [18]. Typestates can be regarded as JML abstract variables and, therefore, JML tools can be used to simulate the typestate verification of specifications. However, JML does not provide support for the reasoning about access permissions and JML's support for concurrency is rather limited. The work presented here is more complex than the work in [17], as it involves reasoning about the concurrency properties of a system. A similar direction is taken in [19] with a formal framework to model-check JML specification using the Bogor model-checker. The Plural group has conducted several case studies to verify API protocols using DFA techniques [7]. By

contrast, our technique is able to analyze the specification for any possible concurrent execution of programs implementing it, while the DFA analysis of Plural is designed to study one program at a time. In [11], DeLine and Fähndrich use the Fugue protocol checker on a relatively large .Net web based application. Like Plural, Fugue provides support for typestate verification, however, reasoning is provided only in the case of more restrictive **Unique** permissions.

Related techniques for formal verification of specifications include: In [20], the requirements for the TCAS airborne, collision avoidance protocol formulated in RSML were checked with SMV. The model checker builds a highly abstract model to avoid the state-space explosion problem. The TLA$^+$ specification of a Compaq multiprocessor cache coherence protocol was verified in TLC to verify design. An "everything is a set" approach to translating Z into SAL is presented in [21], but not fully automated and applied only to small models. The Alloy analyser [22] supports a very expressive language, but is not a temporal logic model checker. ProB [23] is an animator and model checker for B specifications that can detect deadlocks and invariant violations. The Verifast tool [24] provides support for verifying fractional permissions in a similar fashion to Plural. Validating temporal properties of software has been proposed in [25] and applied to Windows NT drivers. The technique, based on predicate abstraction, is implemented in the SLAM toolkit. In [26], the Vault programming language is used to describe resource management protocols that the compiler can statically enforce through a certain order of operations for a given data object. In [27], the Bandera Specication Language (BSL) based on assertions and pre/post conditions of methods, is translated into the input of several model checkers such as Spin and NuSMV, to verify a variety of system correctness properties. In [28], a more expressive LTL$^e$ is proposed to verify high-level specification languages such as B, Z and CSP. The model checker based on extended LTL$^e$ can query deadlock and partially explored state spaces in an effective way. In a slightly recent work [29], a small specification language PL that models business process, is translated into linear temporal formulas, to check the deadlock freedom of interacting business processes of an airline ticket reservation system. In another work [30], We have also used symbolic model checking to check the locking mechanism of the Linux Virtual File System (VFS) by extracting abstract models from the Linux kernel. Our work analyses specifications based on access permissions and typestates, that according to best knowledge of authors is not the subject of previous work. We apply our technique on a relatively large set of MTTS specifications. The generated model also does not require any pre-processing overhead, due to less syntactic sugar.

## 8. Conclusion and Future Work

The MTTS is a relatively large size commercial application that implements a server with a thread pool that runs processing tasks. The specification and verification of the MTTS was a challenging due to the scale and complexity of the application. Overall, we specified and verified forty nine Java classes with 14451 lines of Java code and 546 lines of program specifications written in Plural. Plural takes between a couple of milliseconds for the verification of small classes to a couple of minutes for the verification of large classes. The code of the MTTS was not originally documented so

we needed to document it prior to its specification and verification. We kept the code of the MTTS unchanged as much as possible and tried to keep the semantics of the code intact whenever we introduced any changes to it.

This is the first case study on the use of Plural for the verification of a commercial large sized application. Some of the limitations of Plural (see discussion in Section 3.5 for a full list) hampered our specification and verification work. Nonetheless, we managed to specify and verify important design properties backing the implementation of the MTTS application. The written typestate specifications can further be used to generate a collection of documents describing the behaviour of the MTTS, which can be used for the quality assurance team of Novabase for different purposes. Our experience provides evidence that typestate and permission-based verification tools, such as Plural, can be practically applied to commercial software and can assure important properties of the source code.

In order to overcome some of Plural's limitations, particularly with respect to ensuring that the specification is valid, we developed the Pulse tool. Pulse was able to identify over a hundred previously unknown issues in the MTTS specification in the form of unreachable states and methods. In addition, Pulse was able to show the absence of deadlock states. This experience suggests that Pulse can be a useful tool in checking the consistency of specifications based on typestate and permissions.

Our study suggested a number of interesting directions for future work. As writing program specifications for medium sized or large applications can be a laborious and sometimes complex task, we are investigating inference of Plural specifications. Our study showed that particularly in concurrent applications, it's useful to keep track of intermediate states that an object may go through while the object is executing. Plural could be improved by an explicit specification construction indicating transition into and out of these intermediate states. Finally, in the Æminium project [31] we are investigating leveraging the permissions used in Plural not just for verification, but also to parallelize code. Through further work in the area, we believe permissions will prove to provide a solid foundation for specification, verification, and implementation of robust concurrent software in the future.

## References


[1] J. Boyland, J. Noble, W. Retert, Capabilities for sharing: A generalisation of uniqueness and read-only, in: Proc. 15th European Conference on Object-Oriented Programming, ECOOP, Springer-Verlag, London, U.K., 2001, pp. 2–27.

[2] J. Boyland, Checking interference with fractional permissions, in: Proceedings of the 10th International Conference on Static analysis, SAS, 2003, pp. 55–72.

[3] C. Boyapati, R. Lee, M. C. Rinard, Ownership types for safe programming: preventing data races and deadlocks, in: Proceedings of International Conference on Object-Oriented Programming, Systems and Applications, OOPSALA, 2002, pp. 211–230.

[4] J. Vitek, B. Bokowski, Confined types, in: Proceedings of conference on Object-Oriented Programming Systems and Applications, OOPSALA, 1999, pp. 82–96.

[5] The Plural Tool, http://code.google.com/p/pluralism/.

[6] I. Ahmed, N. Cataño, R. Siminiceanu, J. Aldrich, The Pulse tool, http://poporo.uma.pt/~ncatano/Projects/aeminium/pulsepulse/pulse.php (2011).

[7] K. Bierhoff, J. Aldrich, Modular typestate checking of aliased objects, in: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-Oriented Programming Systems and Applications, OOPSLA, 2007, pp. 301–320.

[8] R. E. Strom, S. Yemini, Typestate: A programming language concept for enhancing software reliability, IEEE Transactions on Software Engineering (TSE) 12 (1) (1986) pp. 157–171.

[9] P. Roux, R. Siminiceanu, Model checking with edge-valued decision diagrams, in: NASA Formal Methods Symposium (NFM), NASA/CP-2010-216215, NASA, Langley Research Center, 2010, pp. 222–226.

[10] L. de Moura, S. Owre, N. Shankar, The SAL Language Manual, Tech. Rep. SRI-CSL-01-02, CSL Technical Report (2003).

[11] R. DeLine, M. Fähndrich, The Fugue protocol checker: Is your software baroque?, Tech. Rep. MSR-TR-2004-07, Microsoft Research (Jan. 2004).

[12] J.-Y. Girard, Linear logic, Theoretical Computer Science 50 (1) (1987) pp. 1–101.

[13] I. Ahmed, N. Cataño, Architecture of Novabase' MTTS application, Tech. rep., The University of Madeira, http://www3.uma.pt/ncatano/aeminium/-Documents_files/mtts.pdf (2010).

[14] C.-B. Breunesse, N. Cataño, M. Huisman, B. Jacobs, Formal methods for smart cards: An experience report, Science of Computer Programming 55 (1-3) (2005) 53–80.

[15] N. Cataño, M. Huisman, Formal specification of Gemplus' electronic purse case study, in: L.-H. Eriksson, P. A. Lindsay (Eds.), Formal Methods Europe (FME), Vol. 2391 of Lecture Notes in Computer Science, Springer-Verlag, Copenhagen, Denmark, 2002, pp. 272–289.

[16] E. M. Clarke, E. A. Emerson, A. P. Sistla, Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications, ACM Transaction on Programming Languages and Systems (TOPLAS) 8 (2) (1986) pp. 244–263.

[17] N. Cataño, T. Wahls, Executing JML specifications of Java Card applications: A case study, in: 24th ACM Symposium on Applied Computing, Software Engineering Track, SAC-SE, Honolulu, Hawaii, 2009, pp. 404–408.

[18] G. Leavens, A. Baker, C. Ruby, Preliminary design of JML: A behavioral interface specification language for Java, Software Engineering Symposium (SIGSOFT) 31 (3) (2006) pp. 1–38.

[19] Robby, E. Rodríguez, M. B. Dwyer, J. Hatcliff, Checking jml specifications using an extensible software model checking framework, STTT 8 (3) (2006) 280–299.

[20] M. P. E. Heimdahl, N. G. Leveson, Completeness and consistency in hierarchical state-based requirements, IEEE Transactions on Software Engineering 22 (6) (1996) 363–377.

[21] G. Smith, L. Wildman, Model checking Z specifications using SAL, in: 4th International Conference of B and Z Users (ZB), Vol. 3455 of Lecture Notes in Computer Science, 2005, pp. 85–103.

[22] D. Jackson, Alloy: Lightweight object modelling notation, ACM Transactions on Software Engineering and Methodology 11 (2) (2002) 256–290.

[23] D. Plagge, M. Leuschel, Validating Z specifications using the ProB animator and model checker, in: Integrated Formal Methods, Vol. 4591 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2007, pp. 480–500.

[24] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, F. Piessens, Verifast: a powerful, sound, predictable, fast verifier for C and Java, in: Proceedings of the Third international conference on NASA Formal methods, NFM, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 41–55.

[25] T. Ball, S. K. Rajamani, Automatically validating temporal safety properties of interfaces, in: Proceedings of the 8th international Workshop on Model checking Software, SPIN, 2001, pp. 103–122.

[26] R. DeLine, M. Fähndrich, Enforcing high-level protocols in low-level software, in: Proceedings of the ACM SIGPLAN 2001 conference on Programming Language Design and Implementation, PLDI, 2001, pp. 59–69.

[27] J. C. Corbett, M. B. Dwyer, J. Hatcliff, Robby, Expressing checkable properties of dynamic systems: the bandera specification language, International Journal on Software Tools for Technology Transfer (STTT) 4 (2002) 34–56.

[28] D. Plagge, M. Leuschel, Seven at one stroke: LTL model checking for high-level specifications in B, Z, CSP, and more, International Journal on Software Tools for Technology Transfer (STTT) 12 (1) (2010) 9–21.

[29] P. Y. H. Wong, J. Gibbons, Property specifications for workflow modelling, Science of Computer Programming 76 (10) (2011) 942–967.

[30] A. Galloway, G. Lüttgen, J. Mühlberg, R. Siminiceanu, Model-Checking the Linux Virtual File System, in: Verification, Model Checking, and Abstract Interpretation (VMCAI), Savannah, GA, Vol. 5403 of Lecture Notes in Computer Science, 2009, pp. 74–88.

[31] S. Stork, P. Marques, J. Aldrich, Concurrency by default: using permissions to express dataflow in stateful programs, in: Proceedings of Onward!, 2009, pp. 933–940.