Electronic Communications of the EASST
Volume 53 (2012)

Proceedings of the 12th International Workshop on Automated Verification of Critical Systems (AVoCS 2012)

Automated Verification of Specifications with Typestates and Access Permissions

Radu I. Siminiceanu Ijaz Ahmed and Néstor Cataño

15 pages

Guest Editors: Gerald Lüttgen, Stephan Merz
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Automated Verification of Specifications with Typestates and Access Permissions

Radu I. Siminiceanu[1] * **Ijaz Ahmed**[2] † **and Néstor Cataño**[3] †

[1] radu@nianet.org National Institute of Aerospace, Hampton, Virginia 23666, USA
[2] ijaz.ahmed@m-iti.org The University of Madeira, Madeira ITI
[3] ncatano@uma.pt Carnegie Mellon University - Portugal, Madeira ITI

Abstract:

We propose an approach to formally verify Plural specifications of concurrent programs based on access permissions and typestates, by model-checking automatically generated abstract state-machines. Our approach captures all possible relevant behaviors of abstract concurrent programs implementing the specification. We describe the formal methodology employed in our technique and provide an example as proof of concept for the state-machine construction rules. We implemented the fully automated algorithm to generate and verify models as a freely available plug-in of the Plural tool, called Pulse. We tested Pulse on the full specification of a Multi Threaded Task Server commercial application and showed that this approach scales well and is efficient in finding errors in specifications that could not be previously detected with the Data Flow Analysis (DFA) capabilities of Plural.

Keywords: Typestates, Access Permissions, Program Specifications, Model-Checking, Concurrency.

1 Introduction

The idea of model checking specifications has been around for more than two decades [HJL96, Jac94]. However, after an initial wave of breakthroughs in the mid 1990s, the enthusiasm has significantly cooled off. This is due to the tacit understanding of the fact that fully automated verification of formal specifications is an elusive goal, due to the disproportionate gap between the expressive power of rich specification languages and the simplicity of temporal logics used by model-checkers. Theorem provers and satisfiability solvers may succeed in this endeavour, but it is generally accepted that one has to trade expressiveness for gains in decidability. Alternatively, one may choose to model certain aspects of the specification that can be tackled by model checking via sound abstractions. These facts have not visibly changed in the last decade.

In this paper we follow the latter approach, by applying an efficient symbolic model checking technique to a selected set of aspects of the specification language used by Plural [Plu] to perform an exhaustive analysis of properties related to concurrency. We implemented the proposed

* This work has been supported by NASA Cooperative Agreement NNX08AC59A, subagreement number 27-001310.
† 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.

model-checking techniques as the Pulse [ACSA11] tool. The Plural specification language is based on *access permissions* and *typestates*. Access permissions are abstract descriptions of how various references to an object can coexist [BNR01]. For instance, a *unique* access permission describes the case when only one reference to a particular object exists, thus enforcing absence of interference and enabling parallelization of code. Typestates enable the tracking of syntactically correct but semantically undefined sequences of program instructions by defining the correct object (type-) states on which they can be called [SY86].

The analysis performed by Pulse focuses on a core set of guarantees related to concurrency: absence of deadlock, absence of unreachable code, the typestate reachability graph has the intended structure, and access permissions are not used incorrectly. In contrast to the existing DFA analysis in Plural, our model-checking approach verifies specifications and not their program implementation.

The approach to verification presented in this paper is not limited to Plural specifications alone, but may be employed for any language that can capture the concepts of typestates, access permissions and concurrency, such as the language of the Fugue checker [DF04]. More concretely, the contributions of the proposed work are: (1) an exhaustive yet tractable approach to analyze specifications at an appropriate level of abstraction; (2) a discrete semantics for *fractional* access permission manipulation that is scalable; (3) a translation algorithm that allows full automation; (4) an integration of model checking techniques into the Plural framework; (5) the Pulse plug-in.

The paper is organised as follows. Section 2 introduces Plural. Section 3 presents the Pulse tool that implements the model-checking analysis of Plural specifications. Section 4 presents the state-machine model of Plural specifications. Section 5 describes the algorithm used to translate Plural specifications into the state-machine model. Section 6 presents experimental results on Pulse. Section 7 presents related work, and Section 8 presents future directions for our work.

2 Preliminaries

2.1 The Plural Tool

Plural is a sound modular typestate and access permission checker tool implemented as an Eclipse plug-in. The Plural tool takes a Java program annotated with Plural specifications and checks whether the program complies with its specifications or not by using DFA (Diagram Flow Analysis). Plural performs several types of program analysis, e.g. fractional analysis (influenced by Boyland's work in [Boy03]), whereby access permissions can be split in several more *relaxed* permissions and then joined back again to form more *restrictive* permissions. Plural also has a simple effects analyser that checks whether a particular method has side effects or not, and an annotation analysis tool that checks whether annotations are well formed.

There are two main issues stemming from the style of verification of Plural: (1) only one client program (a program that uses the Plural's specified libraries) at a time can be verified, similar to testing, which is not exhaustive; if other errors are manifested by a different client program (in a sequence of method calls), they will not be exposed; (2) if the specification itself has errors or unintended semantics, the programmer might never become aware of it. The model-checking technique presented in this paper can solve some of these issues. Equally important from the practical point of view, the implementation of the technique as the Pulse tool [ACSA11] has

This reference	Other references
Unique	∅
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 [BA07]

scaled well in practice.

2.2 The Plural Specification Language

The Plural specification language combines *access permissions* and *typestates* specifications. Access permissions are abstract capabilities allowing a method to access a particular object state [BNR01]. Plural uses access permissions to keep track of the various references to a particular object, and to check the types of access these references have. Access can be read or write (modify). Typestates define protocols on finite state machines [SY86] as the sets of valid object states on which a method can be called.

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 [Gir87], 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.

Plural provides support for five types of access permissions, namely, **Unique**, **Immutable**, **Full**, **Share**, and **Pure**. **Unique(x)** guarantees that reference **x** is the sole reference to the referenced object; **Immutable(x)** provides **x** with read-only access to the referenced object and allows other references to exist and read from it; **Full(x)** states that **x** has reading and writing access to the referenced object, but other references cannot modify it; **Share(x)** is similar to **Full(x)** except that other references to the object can modify it; **Pure(x)** provides **x** with read-only access to the referenced object and allows the existence of other references to the same object with either read-only or read-and-modify access. Figure 1 presents a taxonomy of the rights an access permission can have and how different access permissions can coexist.

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),

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

```
@Refine ({
  @dim(name="STATS", value={"Filled","Empty"}),
  @dim(name="MUTEX", value= {"Acq","NestedAcq","NotAcq"})
})
@ClassStates({
  @State(name="Acq", inv="own!=null*nested==false"),
  @State(name="NestedAcq", inv="own!=null*nested==true"),
  @State(name="NotAcq", inv="own==null*nested==false"),
  @State(name="Empty", inv="own==null*stat==null"),
  @State(name="Filled", inv="own!=nul*stat!=null")
})
public class MutexImpl extends Mutex {
 private Thread own;
 private boolean nested;
 private MutexStatisticsImpl stat;

 @Perm(ensures = "Unique(this) in Empty*Unique(this) in NotAcq")
 MutexImpl() {}

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

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

 @Perm(requires="Pure(this)")
 public boolean isMutexAcquire() {}

 @Perm(requires="Full(this) in Empty * Full(st) in Filled",
       ensures="Full(this) in Filled")
 public void setStatistics(MutexStatistics st) {}

 @Perm(requires="Pure(this) in Filled", ensures="Pure(this) in Filled")
 public MutexStatistics getStatistics() {}

 @Full(requires="Full(this) in Filled", ensures = "Full (this) in Empty")
 public void resetStatistics() {}
}
```

Figure 2: Specification of class MutexImpl

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 shows the Plural specification of a MutexImpl class taken from a Multi-Task Server Application (MTTS) case study presented in [CA11]. The MutexImpl class implements a reentrant mutex algorithm that handles the access to a critical section. We model two dimensions STATS and MUTEX. The former dimension comprises two typestates Filled and Empty modeling whether any statistical locking information is available or not. The latter dimension models how the object lock is currently acquired: acquired, acquired several times by the same object, or not acquired. Each typestate is defined by its own invariant that relates the typestate with code. Class MutexImpl declares three variables namely own, the owner of the lock, stat, the statistical locking information, and the boolean variable nested that models whether the lock has been acquired several times by the same object.

The class constructor produces a **Unique** access permission for an object that is in state Empty for the dimension STATS, and NotAcq for the dimension MUTEX. The specification for method acquire requires the object to have **Full** access permission and produces the same permissions. The typestate transitions, in each case, from NotAcq to Acq, from Acq to NestedAcq, and from NestedAcq to NestedAcq. The method isMutexAcquire requires **this** to have a **Pure** (read-only) access permission. The method resetStatistics requires **this** to be in typestate Filled with **Full** access permission, and ensures that **this** will be in typestate Empty with **Full** access permission when the method ends.

3 The Pulse Tool

The Pulse tool [ACSA11] takes a Plural annotated Java specification and produces an abstract state machine model expressed in the input language of the `evmdd-smc` model-checker[2]. We use the `evmdd-smc` symbolic model checker [3] [RS10] 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 [RS10] 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. The ability to express custom temporal logic properties for concurrent programs gives `evmdd-smc` further freedom to perform verification tasks tailored to each application.

The `evmdd-smc` state machine model defines three main sections that encode Plural specifications. (1) The first section "variable declaration (initialisation)" declares variables to represent typestates, methods, access permissions and method execution status; (2) the second section "transitions" models method specifications, e.g. pre- and post-conditions; (3) the "properties" section encodes CTL formulae that are verified by the `evmdd-smc` model-checker[4].

We implemented Pulse as an Eclipse plug-in that works on top of the Plural tool. Pulse uses an abstract syntax tree visitor that traverses the Java classes and methods declaration and feeds

[2] Sections 4 and 5 give full details on how these models are generated.
[3] Available at `http://research.nianet.org/~radu/evmdd/`.
[4] Section 5.4 presents the CTL encoding of the properties that Pulse verifies.

relevant information to an ANTLR parser. The ANTLR parser relies on the Plural grammar to process the Plural specifications and to produce the target `evmdd-smc` model. Then, Pulse invokes the `evmdd-smc` model checker to verify properties about the specifications. Pulse processes the `evmdd-smc` output and generates a comprehensive report of the results as a Latex file for the user [5].

4 Abstract Models of Specifications

A Plural specification comprises a finite set of class declarations $C = \{C_1, \ldots, C_c\}$. Every class C_i might contain a set of dimensions declarations $DS_i = \{d_i^1, \ldots, d_i^{g_i}\}$ where g_i is the number of dimensions of the class C_i. The class C_i might also contain typestate declarations $TS_i = \{t_i^1, \ldots, t_i^{h_i}\}$, where h_i is the number of typestates of class C_i, for $1 \leq i \leq c$. The typestate might declare a class invariant that links the typestate with additional constraints. For each typestate dimension, we create an object in the model, and for each class declaration $C_i, 1 \leq i \leq c$, we create a finite number of $K+1$ references to objects of that type: $R_i = \{r_i^0, r_i^1, \ldots, r_i^j, \ldots, r_i^K\}$. 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, a K number of 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 would be allowed for invariant definitions of typestates, 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 work, we do not include integer arithmetic in the invariant expressions, a restriction that allows the construction of sound abstractions for our approach.

4.1 The Basic Component

The building block of our model is the state-machine of an object reference r_i^j that includes:

1. the program counter, $(pc_i^j) \in PC_i = \{exe, done\} \times \{\bot, M_i^1, \ldots, M_i^{m_i}\}$, two per method. The second element of the pair is the identity of the method. For the first element, the value *exe* represents that the method is currently being executed, whereas the value *done* means that the method has finished its execution.

2. the access permissions associated with r_i^j: a field of enumerated type

   $ap_i^j \in AP = \{\bot, \textbf{Unique}, \textbf{Full}, \textbf{Pure}, \textbf{Immutable}, \textbf{Share}\}$.

3. the typestate associated with r_i^j: $ts_i \in TS_i = \left\{\bot, t_i^1, \ldots, t_i^{h_i}\right\}$

We reserve the symbol $\bot$ for undefined values (for multiple domains: typestates, access permissions, methods).

[5] An online version that generates and checks the model is available at http://poporo.uma.pt/~ncatano/Projects/aeminium/pulsepulse/pulse.php; however the Latex report is only generated by the Eclipse Plugin.

4.2 State Transition Rules

Our model allows a non-deterministic transition from *done*-local-states (i.e. a local states with $pc_i^j = (done, \cdot)$) to any *exe*-local-state (i.e. a local states with $pc_i^j = (exe, \cdot)$) provided that it respects its transition guards. This covers all possible sequences of method calls, which is behaviorally equivalent to placing the reference $(this, o_i)$ in any reachable global context.

The transitions from *done*-local-states to *exe*-local-states are guarded by expressions that capture (1) the required typestate condition of the *exe*-local-state, and (2) the required access permissions. 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 algorithm translates the Plural specifications into the abstract model presented in Section 4 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 references:

$$S = \prod_{i=1}^{c} \left(\left\{ \bot, t_i^1, \ldots, t_i^{h_i} \right\} \times \prod_{j=0}^{K} (PC_i \times AP) \right)$$

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*-state, *to*-state) in the transition relation, where unprimed variables refer to the *from*-state and primed variables to the *to*-state.

The routines *StartMethod* (Algorithm 1) and *EndMethod* (Algorithm 2) build the transition relation. The inputs of these routines are the reference r_i^j, the method m, the global context (represented by the global state s and the global typestate t, where global state s includes access permission ap and program counter pc) 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 reference, typestate, and access permission. The output of the algorithm are the 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. The translation algorithm calls a routine $Compatible(ap_x, ap_y)$, that implements a Boolean function that decides whether the access permissions ap_x and ap_y are *compatible*, more precisely if ap_x can be split or merged to ap_y (splitting and merging rules are provided in detail in the technical report [SC11]). The second routine, $ChangePermission(ap_x, ap_y)$, builds the update formula corresponding to a compatible access permission transformation from ap_x to ap_y. We employ the following types:

$$
\begin{array}{lcl}
\mathit{GlobalTypestate} & = & TS_1 \times \ldots \times TS_c \\
\mathit{LocalState} & = & (PC, \mathit{AccessPermission}) \\
\mathit{GlobalState} & = & \text{Array}[1..c] \text{ of Array } [0..K] \text{ of } \mathit{LocalState} \\
\mathit{Reference} & = & (\mathit{ObjectIdx}, \mathit{AliasIdx}) \\
\mathit{Triple} & = & (\mathit{Reference}, \mathit{Typestate}, \mathit{AccessPermission})
\end{array}
$$

Algorithm 1 for the transition corresponding to starting a method

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

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

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

Algorithm 2 for the transition corresponding to ending a method.

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

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

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

5.1 Access Permissions as Global Invariants

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. Each reference has its own share of resources, represented as a fraction or set of tokens. The references can take a portion (fraction) or all tokens, depending on access' needs and then return them back to the bank. Thus, the total number of tokens for each object is preserved as a global invariant. None, a portion of, or all resources can be used at a moment in time. 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): $fr_i^B + \sum_{j=0}^{K} fr_i^j = 1 \wedge fw_i^B + \sum_{j=0}^{K} fw_i^j = 1$.

5.2 Dimensions and Typestate Invariants

Plural provides limited support to the verification of typestate invariants. Plural can verify invariants on boolean properties, e.g. checking for non-nullness, but cannot verify invariants that involve arithmetic predicates, e.g. x $>$ 0 (where x is an integer field). Our abstract model of specifications (Section 4) adds a boolean variable for each class field, e.g. it adds three boolean variables for the class specification in Figure 2, to represent invariants about the stat, own, and nested class fields.

For typestates described by boolean invariants, e.g. typestate Empty, the *update* formula restricts the underlying variables, e.g. own and stats, to satisfy the invariant.

Dimensions allow objects to be in several typestates at the same time, e.g. in Figure 2, an object of type MutexImpl can be in Filled and Acq, or Empty and NotAcq at the same time. However, these typestates must belong to different dimensions. Our abstract model of specifications creates a different object for each dimension along with its transitions. The created objects are independent of each other, thus providing a way for the model checker to detect violations regarding dimensions. For instance, if an object is in several typestates that belong to the same dimension then, the model-checker will issue a error.

5.3 Inheritance and Class Fields Visibility

Although our model-checking approach only analyzes method specifications and not their implementation, we can model aspects related to program implementation such as inheritance and class fields visibility. Inheritance is simulated by modeling an implicit call to the parent class constructor. For example, in generating the model for class MutexImpl, we first generate the model for the parent Mutex class, and then we merge the model of the parent class constructor with the model of the MutexImpl class constructor in a way that respects the semantics of inheritance in Java. Hence, the model-checker will report any violations found in the specification of the parent class.

We believe that an approach similar to the "merging" constructor can be used to handle nested method calls. However at the moment, the Pulse tool does not handle that. Also, in Java, class fields can have different levels of visibility, e.g. *public*, *protected*, or *private*. Pulse creates an alias for every public class field in the model however we do not create the alias for private class fields.

5.4 Checking Properties

Checking Sink (Deadlock) states. 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 mutual circular wait) is one particular undesired behavior. In CTL, this can be expressed as:

$$deadlock : \neg \mathrm{EX}(true)$$

Satisfiability of Method Preconditions. The unsatisfiability of the requires clause of a method may be caused by the non-availability of access permissions or typestates, hence the method may

never be called by another object. In our model, we represent this through the CTL property below that uses the predicate $satisfiability_i(m_n)$ to check whether the precondition of method m_n is satisfiable.

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

Typestate Transition Matrix. Often when laying out a specification, the designer knows in advance the expected control flow through the typestates of a class, which can be captured by means of the typestate transition matrix very well. The intended matrix can then be compared with the actual one, computed by Pulse. In CTL, this can be expressed as the formula below, where $adjacent_i(t_1, t_2)$ checks whether a transition between t_1 and t_2 exists or not.

$$\forall 1 \leq i \leq c, \forall t_1 \neq t_2 \in TS_i\text{: } adjacent_i(t_1, t_2) : state_i = t_1 \wedge \mathrm{EX}(state_i = t_2)$$

Checking Concurrency. Access permissions can be used to represent parallel executions of methods along with some other dependency information. Parallel execution is expressed by the CTL formula below, where $concurrent_i(m_1, m_2)$ checks whether two methods can be run in parallel, meaning that two program counters for methods m_1 and m_2 exist with a value exe.

$$\begin{gathered}\forall 1 \leq i \leq c, 0 \leq j_1 \neq j_2 \leq K, \forall m_1 \neq m_2 \in M_i \\ concurrent_i(m_1, m_2) : \mathrm{EF}\left(pc_i^{j_1} = (m_1, exe) \wedge pc_i^{j_2} = (m_2, exe)\right)\end{gathered}$$

Correctness of Access Permissions. Integrity checks can be performed to ensure that an access permission does not violate its intended semantics. We enforce access permissions correctness through the model construction presented in Section 5.1. In CTL, this is expressed as:

$$\begin{gathered}\forall 1 \leq i \leq c, \forall m \in M_i, \forall 0 \leq j_1 \neq j_2 \leq K\text{: } not\_unique(m) : \mathrm{EF}\left(pc_i^{j_1} = (m, exe) \wedge ap_i^{j_2} \neq \bot\right) \\ not\_full(m) : \mathrm{EF}\left(pc_i^{j_1} = (m, exe) \wedge pc_i^{j_2} = (\cdot, exe) \wedge tkw_i^{j_2} > 0\right)\end{gathered}$$

6 Experimental Results

Running Pulse on the Plural specifications presented in Figure 2 produces the results shown in Tables 1, 2 and 3. Table 1 shows the results of the satisfiability of the pre-conditions (the "requires" part) of the methods of the MutexImpl class. The results show that the requires clause of all the methods are satisfiable. Table 2 shows the possible transitions among typestates. The symbol $\uparrow$ indicates the possible transition among typestates and the symbol $\times$ indicates that there is no possible transition. For instance, there is a transition from typestate NotAcq to typestate Acq, but there is no transition from NotAcq to NestedAcq. The typestate transition matrix can be used to reason about the control-flow among typestates. Table 3 shows the possible pair of methods that can be executed in parallel or sequentially. The symbol $\parallel$ indicates that the methods can be executed in parallel and the symbol $\nparallel$ indicates that methods cannot be executed in parallel. For instance, no methods can be executed in parallel with the constructor; similarly, methods that requires **Full** (modifying) access permissions, e.g. acquire and release, cannot be

Method	Satisfiability
MutexImpl	√
acquire	√
release	√
isAcquired	√
getStatistics	√
setStatistics	√
resetStatistics	√

Table 1: Satisfiability of Requires Clauses

	Empty	Filled	Acq	NestedAcq	NotAcq
Empty	↑	↑	×	×	×
Filled	↑	↑	×	×	×
Acq	×	×	↑	↑	↑
NestedAcq	×	×	↑	↑	×
NotAcq	×	×	↑	×	↑

Table 2: Typestate Transition Matrix

executed in parallel. However, the methods that requires **Pure** (read-only) access permissions, e.g. isAcquired and getStatistics, can be executed in parallel with each other. We tested Pulse on the specification of the Multi Threaded Task Server (MTTS) commercial application that was subject to a previous case study [CA11]. MTTS organises tasks in queues and schedules threads to run the tasks. MTTS contains four packages: `library`, `mtts`, `il` (intelligent lock) and `server`, totaling 55 classes, 376 methods, 14451 lines of code, and 546 lines of Plural specification. The core aspects of the specification are related to the modularity and design of the MTTS application, and concurrency issues such as mutual exclusion.

Table 4 presents a synopsis of the analysis carried out by Pulse on the MTTS specifications with K=0 (i.e., no concurrency in this case). In the table, SS stands for sink states, SMP for satisfiability of method preconditions and STM for state transition matrix. Columns 5 to 7 show the number of checked properties. Columns 8 to 10 show the time taken by Pulse (in seconds) to perform the tests. The last three columns show the number of violations found, more precisely, the number of sink states, unsatisfiable requires clauses, and unreachable typestates, respectively. 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 complexity (State Space) and time (Runtime).

Pulse found errors in the MTTS specification which were undetected by the Plural DFA analyzer. It found 58 unsatisfiable requires clauses and 60 unreachable typestates in the full model. The main reasons of these violations are (a) the use in specifications of different typestates that belong to the same dimension, (b) the writing of wrong specifications that lead to unreachable typestates or methods, and (c) the specification of class constructors that do not produce access

	MutexImpl	acquire	release	isAcquired	setStatistics	getStatistics	resetStatistics
MutexImpl	∦	∦	∦	∦	∦	∦	∦
acquire	∦	∦	∦	∥	∦	∥	∦
release	∦	∦	∦	∥	∦	∥	∦
isAcquired	∦	∥	∥	∥	∥	∥	∥
setStatistics	∦	∦	∦	∥	∦	∥	∦
getStatistics	∦	∥	∥	∥	∥	∥	∥
resetStatistics	∦	∦	∦	∥	∦	∥	∦

Table 3: Method Concurrency Matrix

Packages	Classes	Methods	State Space	#Properties			Runtime (s)			Violations		
				SS	SMP	STM	SS	MR	STM	SS	SMP	STM
`library`	8	39	1×10^8	1	39	6	0.07	0.30	0.04	0	0	1
`il`	13	61	7×10^9	1	61	96	0.10	0.18	0.09	0	0	9
`mtts`	19	166	2×10^{16}	1	166	98	0.11	0.33	0.17	0	44	26
`il`, `library`	21	100	1×10^{18}	1	100	102	0.08	0.25	0.15	0	0	27
`il`, `library`, `mtts`	40	266	2×10^{34}	1	266	200	0.43	0.89	0.44	0	44	37
`il`, `library`, `mtts`, `server`	55	368	8×10^{52}	1	368	280	24.34	152.39	2824.47	0	58	60

Table 4: Results: using Pulse on the MTTS specification.

permissions. Wrongly specified constructors leads to a situation in which class method preconditions are never satisfied.

7 Related Work

In [HL96], 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 [SW05], but not fully automated and applied only to small models. The Alloy analyser [Jac02] supports a very expressive language, but is not a temporal logic model checker. ProB [PL07] is an animator and model checker for B specifications that can detect deadlocks and invariant violations. The Verifast tool [JSP+11] provides support for verifying fractional permissions in a similar fashion to Plural. Validating temporal properties of software has been proposed in [BR01] and applied to Windows NT drivers. The technique, based on predicate abstraction, is implemented in the SLAM toolkit. In [DF01], the Vault pro-

gramming 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 [CDHR02], 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, so as to verify a variety of system correctness properties. In [PL10], 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 detect deadlocks and partially explored state spaces in an effective way. In a recent work [WG11], a small specification language PL that models business process, is translated into linear temporal formulas, to check deadlock freedom of interacting business processes of an airline ticket reservation system. 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 does not require any pre-processing overhead, due to less syntactic sugar.

The Plural group has conducted several case studies to verify API protocols using DFA techniques [BA07]. 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. We have also used symbolic model checking in [GLMS09] to check the locking mechanism of the Linux Virtual File System (VFS) by extracting abstract models from the Linux kernel.

8 Conclusion and Future Work

The work presented in this paper translates Plural specifications into evmdd-smc models. This opens up a new window for the evaluation of Plural specifications. The main challenge is to encode the access permissions and dimensions into the model.

The Pulse tool was used to detect a significant number of errors in the MTTS specification [CA11] that were previously undetected by the Plural DFA analyzer. The relatively high number of detected errors shows that model checking of specifications can also be practical. Our approach is not limited to Plural specifications alone, but can be customized for other specifications languages (and even for radical new programming languages like Plaid[6]) based on typestates and concurrency. The approach is particularly useful for evaluating reusable libraries that can have multiple clients. The designer of the libraries can reason about all possible calling scenarios for the methods by using the State Transition Matrix. Pulse helps the designer to investigate the behavior of the library objects in the presence of multiple references to the same object. The experimental results on Pulse suggest that our approach is reasonably scalable.

There are several extensions possible for our approach. One potential avenue is to avoid explicitly encoding the access permissions into the states, but instead use a deductive method (SMT solver) to "decide" if one can transition from a particular pre-access permission to a particular post-access permission. We would also like to explore the possibility of representing access permission fractions explicitly in a model, which would therefore require abandoning the traditional model checking framework, that only uses discrete-state systems, and using more powerful, deductive techniques. The current semantics of **Full** (write) access permission is the generic ability

[6] http://www.cs.cmu.edu/ aldrich/plaid/plaid-intro.pdf

to "modify" the object, which potentially includes the permission to change the size (or even delete it – possible in C++, not in Java). There are collateral implications in allowing the deletion of the object. Therefore the *write* permission can be further refined into two subcategories of access: *modify-but-not-delete* and *delete*.

Bibliography

[ACSA11] I. Ahmed, N. Cataño, R. Siminiceanu, J. Aldrich. The Pulse Tool. 2011. http://poporo.uma.pt/~ncatano/Projects/aeminium/pulsepulse/pulse.php [Online].

[BA07] 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, pp. 301–320. 2007.

[BNR01] J. Boyland, J. Noble, W. Retert. Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only. In *Proceedings of the 15th European Conference on Object-Oriented Programming*. ECOOP, pp. 2–27. Springer-Verlag, London, U.K., 2001.

[Boy03] J. Boyland. Checking interference with fractional permissions. In *Proceedings of the 10th International Conference on Static analysis*. SAS, pp. 55–72. 2003.

[BR01] T. Ball, S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th international Workshop on Model checking Software*. SPIN, pp. 103–122. 2001.

[CA11] N. Cataño, I. Ahmed. Lightweight Verification of a Multi-Task Threaded Server: A Case Study With The Plural Tool. In *Formal Methods for Industrial Critical Systems (FMICS)*. Lecture Notes in Computer Science 6959, pp. 6–20. Trento, Italy, 2011.

[CDHR02] 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)* 4:34–56, 2002.

[DF01] 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, pp. 59–69. 2001.

[DF04] R. DeLine, M. Fähndrich. The Fugue Protocol Checker: Is Your Software Baroque? Technical report MSR-TR-2004-07, Microsoft Research, Jan. 2004.

[Gir87] J.-Y. Girard. Linear Logic. *Theoretical Computer Science* 50(1):pp. 1–101, 1987.

[GLMS09] 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*. Lecture Notes in Computer Science 5403, pp. 74–88. 2009.

[HJL96] C. L. Heitmeyer, R. D. Jeffords, B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodoly (TOSEM)* 5(3):231–261, July 1996.

[HL96] M. P. E. Heimdahl, N. G. Leveson. Completeness and Consistency in Hierarchical State-Based Requirements. *IEEE Transactions on Software Engineering* 22(6):363–377, June 1996.

[Jac94] D. Jackson. Abstract Model Checking of Infinite Specifications. In *Proceedings of Formal Methods Europe*. FME, pp. 519–531. Springer-Verlag, London, U.K., 1994.

[Jac02] D. Jackson. Alloy: Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology* 11(2):256–290, Apr. 2002.

[JSP+11] 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, pp. 41–55. Springer-Verlag, Berlin, Heidelberg, 2011.

[PL07] D. Plagge, M. Leuschel. Validating Z Specifications Using the ProB Animator and Model Checker. In *Integrated Formal Methods*. Lecture Notes in Computer Science 4591, pp. 480–500. Springer Berlin / Heidelberg, 2007.

[PL10] 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):9–21, Jan. 2010.

[Plu] The Plural Tool. http://code.google.com/p/pluralism/ [Online].

[RS10] P. Roux, R. Siminiceanu. Model Checking with Edge-valued Decision Diagrams. In *NASA Formal Methods Symposium (NFM), NASA/CP-2010-216215*. Pp. 222–226. Langley Research Center, April 2010.

[SC11] R. Siminiceanu, N. Cataño. Automated Verification of Specifications with Typestates and Access Permissions. Technical report NASA/CR-2011-217170 NF1676L-13249, NASA Langley Research Center, August 2011.

[SW05] G. Smith, L. Wildman. Model Checking Z Specifications Using SAL. In *4th International Conference of B and Z Users (ZB)*. Lecture Notes in Computer Science 3455, pp. 85–103. 2005.

[SY86] R. E. Strom, S. Yemini. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Transactions on Software Engineering (TSE)* 12(1):pp. 157–171, January 1986.

[WG11] P. Y. H. Wong, J. Gibbons. Property specifications for workflow modelling. *Science of Computer Programming* 76(10):942–967, October 2011.