NASA/CR–2011-217170

Automated Verification of Specifications With Typestates and Access Permissions

Radu I. Siminiceanu
National Institute of Aerospace, Hampton, Virginia

Néstor Cataño
Madeira ITI, CMU-Portugal, Campus da Penteada, Funchal, Portugal

August 2011

NASA/CR–2011-217170

Automated Verification of Specifications With Typestates and Access Permissions

Radu I. Siminiceanu
National Institute of Aerospace, Hampton, Virginia

Néstor Cataño
Madeira ITI, CMU-Portugal, Campus da Penteada, Funchal, Portugal

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Cooperative Agreement NNX08AC59A

August 2011

Acknowledgments

This work has been supported by NASA Cooperative Agreement NNX08AC59A subagreement number 27-001310 and by CMU-Portugal under project grant CMU-PT/SE/0038/2008.

The use of trademarks or names of manufacturers in this report is for accurate reporting and does not constitute an offical endorsement, either expressed or implied, of such products or manufacturers by the National Aeronautics and Space Administration.

Available from:

NASA Center for AeroSpace Information
7115 Standard Drive
Hanover, MD 21076-1320
443-757-5802

Abstract

We propose an approach to formally verify Plural specifications based on access permissions and typestates, by model-checking automatically generated abstract state-machines. Our exhaustive approach captures all the possible behaviors of abstract concurrent programs implementing the specification. We describe the formal methodology employed by our technique and provide an example as proof of concept for the state-machine construction rules. The implementation of a fully automated algorithm to generate and verify models, currently underway, provides model checking support for the Plural tool, which currently supports only program verification via data flow analysis (DFA).

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. For example, the Æminium research project [19], which succeeds Plural [15], is developing a platform that enables programmers to write *concurrent-by-default* programs, and is prone to support massive parallelism. The Plural language uses Java syntax and is inspired by purely functional programming languages in which programmers express not a sequence of side-effect operations but rather how a result should be built functionally from an input. The order of the execution does not matter as long as the dependencies within the program are respected. The Plural language proposes a new way of structuring object oriented programming based on code dependencies and data-flow alone. Plural programmers provide data dependencies thus enabling Plural runtime system to parallelize the program. Dependencies are written as *access permissions* [3, 6] that express logical dependencies on concurrent execution runs. Access permissions are abstractions describing how objects are accessed. For instance, a **Unique** access permission describes the case when a sole reference to a particular object exists, and a **Shared** access permission models the case when an object is accessed by multiple references. Access permissions are inspired by Girard's Linear Logic [12]. Method access permissions specifications are produced and consumed, in the spirit of Linear Logic.

The Æminium concurrent-by-default platform is still under development and efforts are in place to develop techniques to improve the analysis performed by its current implementation. In particular, Æminium does not provide support for model checking. In this paper, we propose a new formal approach that enables reachability analysis of Plural specifications. First, we translate Plural specifications into abstract state-machines that capture all possible behaviors allowed by the specification. Then, we use a symbolic model checker based on edged-valued decision diagrams (EVMDD) [16] to verify that specifications satisfy a set of basic integrity properties such as absence of deadlock, absence of unreachable code, and correct use of access permissions. Additionally, the ability to express custom temporal logic properties for concurrent programs gives the analysis the freedom to perform verification tasks tailored to each application. Finally, we use a Petri net inspired semantics to represent access permission transformations. In contrast to Plural, the focus of our analysis is not on program verification (concrete code implementing a specification) but on verifying the specification itself.

The rest of this document is organized as follows. The remaining part of this section presents related work. Section 2 introduces the specification language used by Plural (access permissions) and its extension used by the Plural tool (access permissions and typestates). Section 3 presents the state-machine model used for verifying Plural specifications. Section

4 describes the algorithm that translates Plural specifications into the state machine model. Section 5 presents the Petri-Nets based approach to formally verifying Plural specifications, together with a case study taken from a commercial data base application.

Related Work: In previous work [8], we used the `jmle` tool [13] for writing JML specifications [14] of an electronic purse application written in the Java Card dialect of Java. JML is a behavioral interface specification language for Java, which means that the only correct implementation of a JML class specification is a Java class with the specified behavior. JML method specifications are written with the aid of the keywords **requires**, **modifies** and **ensures**, which respectively give the precondition, the frame (what locations may change from the pre- to the post-state) and the postcondition. JML provides support for the declaration and use of specification-only variables (e.g., using the **model** construct). Typestates can be regarded as JML abstract variables and so JML tools can be used to simulate typestate verification of specifications. However, JML does not provide support to the reasoning about access permissions. JML's support for concurrency is rather limited. The work presented here is more complex than the work presented in [8], as it involves reasoning on concurrency properties of a system.

The Plural group has conducted different case studies on the use of typestates to verify Java I/O stream libraries [3], Java iterator libraries [4], and Java database libraries [5]. The main goal of these case studies is to show that Plural can effectively be used to check violations of APIs protocols. The work presented in this document focuses on verifying Plural specifications in isolation regardless of the program source code, instead.

Validating temporal safety properties of software has been proposed in [2] and applied to Windows NT drivers. The technique, based on predicate abstraction, is implemented in the SLAM toolkit. It automatically creates abstractions of C code using iterative refinement and has been used to verify correct locking behavior. In [10], the Vault programming language is used to describe resource management protocols that the compiler can statically enforce. Protocols can specify that certain operations must be performed in a certain order and that certain operations must be performed before accessing a given data object. The technique has been used on the interface between the Windows kernel and its device drivers.

Finally, we have previously employed symbolic model checking for analyzing languages and software. In [11], we have checked the locking mechanism of the Linux Virtual File System (VFS) by extracting abstract models from the Linux kernel. In [18], we have validated the correctness of syntactic macro-definitions proposed for the plan execution language PLEXIL, and in [17] we have studied the semantics of temporal properties specified in the planning language ANMLite.

2 Specifications for Plural Programs

The Plural language is based on *access permissions* and Plural combines these with *typestates*. Typestates define protocols on finite state machines [20]. They can be used as abstractions to reason about objects and programs [1]. They are abstract definitions on the capability of a method to access a particular state [3, 6]. Access permissions are inspired by Girard's Linear Logic [12], hence, they can be used, produced and consumed. Access permissions are used 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). Plural provides support to five types of access permissions, namely, **Unique**, **Share**, **Immutable**, **Full**, and **Pure**. Figure 1 presents a taxonomy of how different access permissions can coexist. For example, **Full** access to a referenced object allows the existence of any other reference with **Pure** access to the same referenced object.

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 [3]

- **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 immutable permissions.

The Linear Logic formula $P \multimap Q$ is modeled as the specification **@Perm**(requires=“**P**”, ensures=“**Q**”) of the Plural language. The semantics of the operator $\otimes$ of Linear Logic, which denotes simultaneous occurrence of resources, is captured by the operator “*”. **P** and **Q** can be a specification such as **Unique**(x) **in A** * **Full**(y) **in B**, which requires (ensures) that reference “x” has **Unique** permission on its referenced object, which should be in state **A**, and simultaneously requires (ensures) that “y” has **Full** permission on its referenced object, which should be in state **B**. The semantics of the additive conjunction operator “&” of Linear Logic, which represents the alternate occurrence of resources, is captured by the use of a **@Cases** specification, the decision of which is made according to a required resource in one of its **@Perm** specifications. The additive disjunction operator $\oplus$ of Linear Logic is modeled by the use of a **@Cases** specification, the decision of which is made according to an ensured resource by one of its **@Perm** specifications. Typestates are declared with the aid of the **@ClassStates** clause. Method specifications are written with the aid of the **@Perm** clause, composed of a “requires” part, describing the conditions required by a method to be executed, and an “ensures” part, describing the conditions that hold after method execution.

Additionally, the clause **@Cases** allows the annotation of several **@Perm** specifications for a method. The constructor of the class ProducerConsumer creates a **Unique** object that is initially in state **Empty**. A **@TrueIndicates** specification defines the state generated by a method when it returns true. The resources are stored in a container of bounded size. Therefore, when the container becomes full, producing is no longer allowed before consuming at least one resource. Consuming is obviously not permitted when the container is empty. The following basic producer/consumer example, with one class, one constructor, two read-only methods, *isEmpty()* and *isFilled()*, and two read/write methods, *produce()* and *consume()*, illustrates an example of a Plural specified program.

```
@ClassStates({
  @State(name = "Empty"),
  @State(name = "Partial"),
  @State(name = "Filled")
})
class ProducerConsumer {
  @Perm(ensures = "Unique(this) in Empty");
  ProducerConsumer() { ... }

  @Pure
  @TrueIndicates("Empty")
  bool isEmpty() { ... }

  @Pure
  @TrueIndicates("Filled")
  bool isFilled() { ... }

  @Cases({
   @Perm(requires = "Full(this) in Empty",   ensures = "Full(this) in Partial"),
   @Perm(requires = "Full(this) in Partial", ensures = "Full(this) in Partial"),
   @Perm(requires = "Full(this) in Partial", ensures = "Full(this) in Filled")
  })
  void produce() { ... }

 @Cases({
  @Perm(requires = "Full(this) in Partial", ensures = "Full(this) in Empty"),
  @Perm(requires = "Full(this) in Partial", ensures = "Full(this) in Partial"),
  @Perm(requires = "Full(this) in Filled",  ensures = "Full(this) in Partial")
 })
 void consume() { ... }
}
```

3 Abstract Models of Plural Specifications

Our first approach to the verification of Plural specifications relies on an algorithm that extracts an abstract state-machine representation of the collective dynamic behaviors of the object references described in the specification. This section introduces the abstract state-machine model, and Section 4 presents the algorithm that translates Plural specifications to the abstract state-machine representation. The main difference between our approach and the line of research pursued by the Plural group in [3,5] is that, while using the same input language, we model and verify the specification alone and not programs (code analysis). Our abstract models are not concerned with the body of methods, but only with their specification: preconditions, postconditions, invariant conditions, and access permissions. Also by contrast, our technique is able to analyze the specification for any possible concurrent execution of programs implementing it, while the Data Flow Analysis (DFA) technique in Plural is designed to study one program at a time.

A Plural specification comprises a finite set of class declarations $\mathcal{C} = \{C_1, \ldots, C_c\}$. Every class C_i contains a set of typestate declarations, $\mathcal{TS}_i = \{t_i^1, \ldots, t_i^{h_i}\}$, where h_i is the number of typestates for class C_i, for $1 \leq i \leq c$. A typestate [20] might declare a class-state invariant that relates the typestate with (Java) code. For each class declaration $C_i, 1 \leq i \leq c$, we create a finite number of instances of references to objects of that type: $\mathcal{R}_i = \{r_i^0, r_i^1, \ldots, r_i^j, \ldots, r_i^K\}$. The parameter K is the only attribute of our model that is determined à priori. A meta-argument is necessary to support the decision to bound K to a predetermined value. We informally argue that K can be set to the number of simultaneous distinct access permissions to the same object, which is 5. The choice of parameter can be validated formally by extending the analysis to $K = 6$ (i.e., by allowing multiple accesses of the same type) and showing that the extended model does not introduce any additional "relevant" behavior of r_i^0 in the model.

We assume a default abstract representation of the operating environment. In a generic reference (x_i, o_i) to an object of class C_i, o_i is the object uniquely described by its physical memory address, and x_i is the name of the alias to the physical object. We reserve r_i^0 to the generic reference (this, o_i). The abstraction represents the congruence relation over the equality with o_i. From the point of view of analyzing the behavior of (this, o_i) in terms of access permissions, the behavior of other objects, (y_h, o_h) with $h \neq i$, is completely independent, since it does not affect o_i. Thus, all references to other physical locations can be captured with a single abstract state, for example $\perp = (\cdot, \mathsf{null})$.

3.1 The Basic Component

The building block of the model generated by our tool is the state-machine of an object reference r_i^j, which includes:

(a) the program counter, $pc_i^j \in \mathcal{PC}_i = \{pre, post\} \times \left(\{\perp\} \cup \{M_i^1, \ldots M_i^{m_i}\}\right)$, two per each method, where we include the constructors in the list of methods. We reserve the symbol $\perp$ for undefined values (for multiple domains: typestates, access permissions, methods), throughout the paper.

(b) the access permissions associated with r_i^j: a field of enumerated type

$ap_i^j \in \mathcal{AP} = \{\perp, \mathsf{Unique}, \mathsf{Full}, \mathsf{Pure}, \mathsf{Immutable}, \mathsf{Share}\}$.

Additionally, for each object o_i we store its typestate (which is shared by all references $r_i^j, 0 \leq j \leq K$) $ts_i \in \mathcal{TS}_i = \{\perp\} \cup \{t_i^1, \ldots, t_i^{h_i}\}$, where $\perp$ describes an "undefined" state of a reference, corresponding to its "pre-creation" state. A reference is created by two means: either by calling a constructor method of C_i or by pointing it to object o_i after its creation (aliasing). As mentioned before, we use the same abstract state $\perp$ to represent references that point to objects other than o_i as well as null references.

An mapping of the variables representing r_i^j to values in their domain is called a *local state*. The cross product of all local states is called a *global state*.

3.2 State transition rules

From each *post*-local-state (a local state with $pc_i^j = (post, \cdot)$) we allow a non-deterministic transition to any other *pre*-local-state. This covers all possible sequences of method calls, which is behaviorly equivalent to placing the reference (this, o_i) in any possible global context. The transitions from *post*-local-states to *pre*-local-states are guarded by expressions that capture several constraints:

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

(ii) the access permission constraints determined by the splitting rules described in subsection 3.3

Additionally, from each *pre*-local-state (pre, m) a reference can only transition to its matching *post*-local-state $(post, m)$, 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 that may occur.

3.3 Access Permissions Splitting Rules

Influenced by Boyland's work in [7], Plural performs *fractional* analysis of access permissions, hence, they can be split into several more *relaxed* permissions and then joined back to form more *restrictive* permissions. Figure 2 presents Plural splitting and joining rules, where at least one of x_1 and x_2 is x, and $k_1 + k_2 = k$. For instance, a Unique access permission of weight k can be split in two $k/2$ Share access permissions. At the present stage of our work, we fully abstract away the notion of permission *fractions*, since traditional model checking techniques do not handle continuous variables.

Splitting rules govern the way transitions from *post*-local-states to *pre*-local-states can be carried out. If the *pre*-local-state of a method requires a reference "x" to have **Unique** permission to an object "o", but the current *post*-local-state ensures that two half **Share** permissions to the same object "o" exist, then the method (non-deterministically) might still be executed.

$$\begin{array}{rcl}
\mathsf{Unique}(x, o, k) & \Longleftrightarrow & \mathsf{Full}(x_1, o, k_1) \otimes \mathsf{Pure}(x_2, o, k_2) \\
\mathsf{Unique}(x, o, k) & \Longleftrightarrow & \mathsf{Share}(x_1, o, k_1) \otimes \mathsf{Share}(x_2, o, k_2) \\
\mathsf{Full}(x, o, k) & \Longleftrightarrow & \mathsf{Immutable}(x_1, o, k_1) \otimes \mathsf{Immutable}(x_2, o, k_2) \\
\mathsf{Immutable}(x, o, k) & \Longleftrightarrow & \mathsf{Pure}(x_1, o, k_1) \otimes \mathsf{Immutable}(x_2, o, k_2) \\
\mathsf{Immutable}(x, o, k) & \Longleftrightarrow & \mathsf{Immutable}(x_1, o, k_1) \otimes \mathsf{Immutable}(x_2, o, k_2)
\end{array}$$

Figure 2. Access permission splitting rules

4 Translation Algorithm

The translation algorithm 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:

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

The transition relation can be defined component-wise, for each reference r_i^j (the local transition relations R_i^j), and the global transition relation is the asynchronous composition of the local transition relations.

Below, define the rules to construct each R_I^j. There are two types of local transitions, corresponding to starting a method and ending a method. In the following, we use the standard notation for pairs of states (*from* states and *to* states) in the transition relation: unprimed variables refer to the *from*-state and primed variables to the *to*-state.

The routines *StartMethod* and *EndMethod* build the transition relation expressions associated with starting and ending ending a method m by a reference r_i^j, respectively. The input for 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), and two triplets. The triplets $\left(r_{i_0}^{j_0}, ts_{i_0}^{k_0}, ap_0\right)$ 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, namely the required and ensured typestate, reference, and access permission. The output of the routines are two Boolean expressions: *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 semantics of this pair of expressions is: "if *guard* evaluates to true in the current state then the transition can be executed and the global state changes according to *update*."

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 computes the guard and update expressions for the transition corresponding to starting a method

```
StartMethod(
  $s$ : $\mathit{GlobalState}$,
  $t$ : $\mathit{GlobalTypestate}$,
  $r_i^j$ : $\mathit{Reference}$,
  $m$ : $\mathit{Method}_i$,
  $\left(\left(r_{i_0}^{j_0}, ts_{i_0}^{k_0}, ap_0\right), \left(r_{i_1}^{j_1}, ts_{i_1}^{k_1}, ap_1\right)\right)$ : $\mathit{Triple} \times \mathit{Triple}$
)
  $\mathit{guard} \leftarrow s[i][j].ap \neq \bot \wedge s[i][j].pc = (\mathsf{post}, \cdot) \wedge t[i_0] = ts_{i_0}^{j_0} \wedge$
      $\mathit{Compatible}\,(s[i_0][j_0].ap, ap_0) \wedge \mathit{Compatible}\,(s[i_1][j_1].ap, ap_1)$
  $\mathit{update} \leftarrow s'[i][j].pc = (\mathsf{pre}, m) \wedge \mathit{ChangePermission}\,(s[i_0][j_0], ap_0)$
  return $\mathit{guard} \Rightarrow \mathit{update}$
```

Algorithm 2 computes the guard and update expressions for the transition corresponding to ending a method.

```
EndMethod(
  $s$ : $\mathit{GlobalState}$,
  $t$ : $\mathit{GlobalTypestate}$,
  $r_i^j$ : $\mathit{Reference}$,
  $m$ : $\mathit{Method}_i$,
  $\left(\left(r_{i_0}^{j_0}, ts_{i_0}^{k_0}, ap_0\right), \left(r_{i_1}^{j_1}, ts_{i_1}^{k_1}, ap_1\right)\right)$ : $\mathit{Triple} \times \mathit{Triple}$
)
  $\mathit{guard} \leftarrow s[i][j].pc = (\mathsf{pre}, m)$
  $\mathit{update} \leftarrow t'[i_1] = ts_{i_1}^{k_1} \wedge s'[i_1][j_1].ap = ap_1 \wedge s'[i][j].pc = (\mathsf{post}, m) \wedge$
      $\mathit{ChangePermission}\,(s[i_1][j_1].ap, ap_1)$
  return $\mathit{guard} \Rightarrow \mathit{update}$
```

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 exact opposite: the object o_i has not been already created.

Integral parts of the algorithms are two routines that require further explanation. The routine *Compatible*(ap_x, ap_y) implements a Boolean function that gives a "true or false" answer to whether the access permissions ap_x and ap_y are "compatible", more precisely if ap_x can be downgraded or upgraded to ap_y according to Section 4.1, Figure 3. The routine *ChangePermission*(ap_x, ap_y) builds the update formula corresponding to a compatible access permission transformation from ap_x to ap_y.

The formal arguments supporting these two routines are described in more detail in the next subsection (4.1). Informally, they implement an abstraction of the access permission joining and splitting rules, without having to explicitly introduce the notion of fractional permissions.

Next AP	**Unique**	**Full**	**Share**	**Immutable**	**Pure**	⊥
Current AP	this others `rw` `rw`	this others `rw` `rw`	this others `rw` `rw`	this others `rw` `rw`	this others `rw` `rw`	this others `rw` `rw`
Unique	↔ == ==	↓ == +=	↓ == ++	↓ =− +=	↓ =− ++	↓ −− ++
Full	↑ == −=	↔ == ==	↓ == =+	↓ =− ==	↓ =− =+	↓ −− =+
Share	↑ == −−	↑ == =−	↔ == ==	⊗ =− =−	↓ =− ==	↓ −− ==
Immutable	↑ =+ −=	↑ =+ ==	⊗ =+ =+	↔ == ==	↓ == =+	↓ −= =+
Pure	↑ =+ −−	↑ =+ =−	↑ =+ ==	↑ == =−	↔ == ==	↓ −= ==
⊥	↑ ++ −−	↑ ++ =−	↑ ++ ==	↑ += =−	↑ += ==	↔ == ==

Figure 3. Access Permission Transformations (downgrade/upgrade).

4.1 Access Permission Compatibility and Transformation Rules

Access permissions can be transformed into a series of more lax permissions or combined together to form a more restrictive permission (see Section 3.3 and Figure 2). Gaining or losing permissions includes "no change" (i.e., the modification does not have to be strict). Scenarios where both **this** and other references strictly gain rights (access permissions) at the same time are considered incompatible. Similarly, scenarios where both **this** and other references strictly lose rights at the same time are also considered incompatible. This roughly follows the principles of linear logic regarding the "preservation" of resources. There are two generic ways of access permission transformations:

- ***Downgrade***: this reference may give up (read/write) rights and other references may gain rights.

- ***Upgrade***: this reference may acquire more rights and other references may lose rights.

Figure 3 describes in an abstract way the implementation of the routines *Compatible*() and *ChangePermission*() presented before. The directional arrows denote the nature of the transformations: ↓ for downgrade, ↑ for upgrade, ↔ for no change, and × for disallowed. Furthermore, the second row of symbols details the read and write permission change for

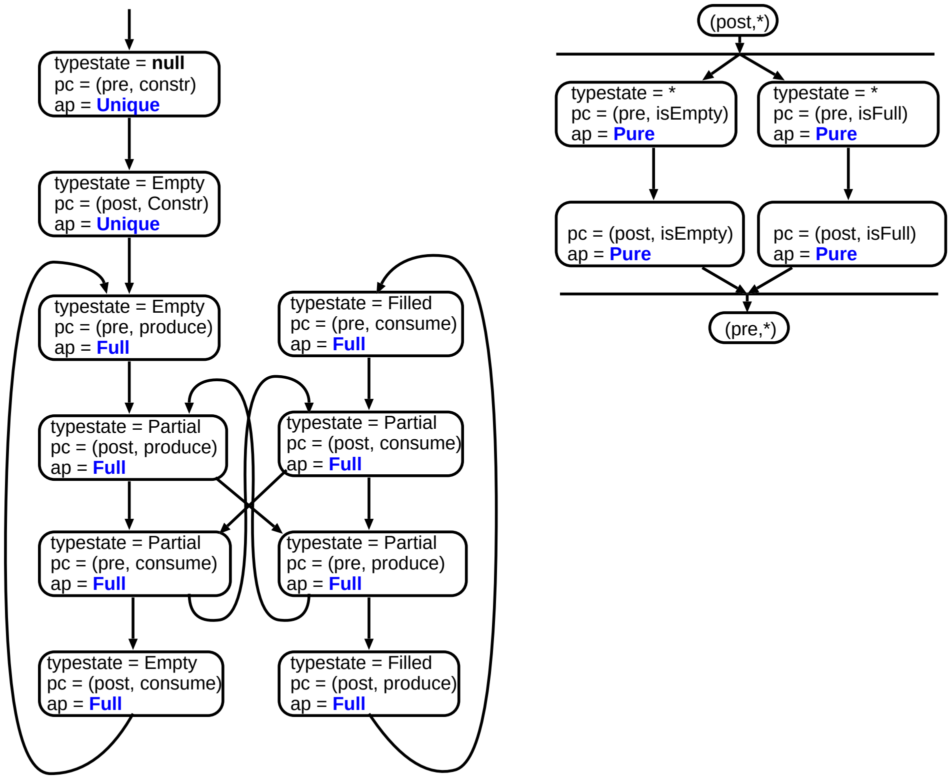

Figure 4. State transition diagram for the producer/consumer example

reference **this** and the other references ("the rest of the world"). Therefore, a **Full** permission is compatible with a **Share** permission. **Full** can be downgraded to **Share** if reference **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.

In the absence of a fully specified constructor, the *default constructor* can be treated as a special pair of triples:

$$((\bot, \mathsf{this}, \bot), (initial, \mathsf{this}, \mathbf{Unique}))$$

Similarly, for creating an alias (assigning a pointer to an existing object):

$$((\bot, \mathsf{this}, \bot), (t[c], \mathsf{this}, \mathbf{Pure}))$$

Figure 4 illustrates the state transition diagram for the basic module of the example of the ProducerConsumer class in Section 2. The transition guards are not depicted for the sake of simplicity, as they can be quite elaborate formulae.

4.2 Experimental Results

We have generated the input for the `evmdd_smc` [16] model checker from the above specification. For $K = 5$, the model has $4,660,153$ states, built in 0.19 seconds on a MacBook. None of the reachable states is deadlocked and the typestate reachability graph is identical to the one in Figure 4, meaning that the specification has no blatant errors. The source code of the model generator and the generated producer/consumer model can be found in the appendix and at `http://www3.uma.pt/ncatano/aeminium/Home.html`. Figure 5 shows the state space size and runtime for checking the generated model parameterized by K (maximum number of coexisting references to the same object).

K	# states	time (sec.)	K	# states	time (sec.)
1	387	0.00	6	44,698,270	0.33
2	4,474	0.01	7	422,650,621	0.52
3	47,301	0.03	8	3,952,714,382	0.74
4	476,422	0.08	10	337,317,139,558	1.32
5	4,660,153	0.19	15	$\sim 2.09 \times 10^{16}$	3.55

Figure 5. Experimental results: scalability for producer/consumer model

5 A Petri Net Inspired Semantics for Fractional Access Permissions

The access permission transformations required by our model are based on an underlying concept of "collective management" of permissions among references to the same object. Intuitively, rights (access permissions) are viewed as resources (e.g., tokens) that are available globally. None, a portion of, or all resources can be used at a moment in time. Each object has its own share of resources, represented as a fraction or set of tokens. For example, a constructor creates an object with exclusive rights (**Unique** permission), therefore the constructor has to request all the tokens for that object. The tokens are initially stored in a "bank", implementing the collective management of permissions, accessible to all. References take and give back tokens depending on access needs, but the total number of tokens for each object is preserved. Hence, a global invariant property is that no resources are created or lost for each individual object.

We can illustrate this concept first with the more general framework of fractional permissions, where fractional permissions add up to 1. Note however, that we use this as a starting analogy only. 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 $\{0, 1, \ldots, K + 1\}$.

A key departure from the original framework is to separate access permissions into read and write and describe the rights 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, respectively) 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 are the unused fractions (still in the bank).

$$fr_i^B + \sum_{j=0}^{K} fr_i^j = 1 \wedge fw_i^B + \sum_{j=0}^{K} fw_i^j = 1$$

The possible combinations of values for fractions is listed in Figure 6. 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.

this	semantic	bank	others
$fr_i^j = 0 \wedge fw_i^j = 0$	null	$fr_i^B \geq 0 \wedge fw_i^B \geq 0$	any
$fr_i^j = 0 \wedge fw_i^j > 0$	no meaning	-	-
$0 < fr_i^j < 1 \wedge fw_i^j = 0$	**Immutable**	$fw_i^B = 1$	$\sum_{l \neq j} fw_i^l = 0$: **Immutable** or **Pure**
$0 < fr_i^j < 1 \wedge fw_i^j = 0$	**Pure**	$fw_i^B < 1$	$\forall l \neq j : fw_i^l \geq 0$, **Pure** or **Full**
$0 < fr_i^j, fw_i^j < 1$	**Share**	$0 \leq fr_i^B < 1 \wedge 0 \leq fw_i^B < 1$	$\forall l \neq j : 0 \leq fr_i^l, fw_i^l < 1$, **Share**
$0 < fr_i^j < 1 \wedge fw_i^j = 1$	**Full**	$0 \leq fr_i^B < 1 \wedge fw_i^B = 0$	$\forall l \neq j : fw_i^l = 0$, **Pure**
$fr_i^j = 1 \wedge fw_i^j = 0$	**Immutable**	$fr_i^B = 0 \wedge fw_i^B = 1$	$\forall l \neq j : fr_i^l = 0 \wedge fw_i^l = 0$, **Null**
$fr_i^j = 1 \wedge fw_i^j = 0$	**Immutable**	$fr_i^B = 0 \wedge fw_i^B < 1$	$\forall l \neq j : fr_i^l = 0 \wedge fw_i^l > 0$, no meaning
$fr_i^j = 1 \wedge 0 < fw_i^j < 1$	no meaning	-	-
$fr_i^j = 1 \wedge fw_i^j = 1$	**Unique**	$fr_i^B = 0 \wedge fw_i^B = 0$	$\forall l \neq j : fr_i^l = 0 \wedge fw_i^l = 0$, null

Figure 6. A fractional permission model

Also note that the nature of others rights can be inferred from the value of **this** reference 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.

With our bounding assumption of a maximum number of K distinct aliases per object, we can implement a simple yet sound system of transformation rules inspired by the aforementioned Petri net semantics. We map the infinite range of fraction values, the dense interval $[0, 1]$, to the discrete domain $\{0, 1, \ldots, K+1\}$, via the abstraction

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

In this model, there is an initial number of $K + 1$ tokens for each object and each access right (read, write) available at the bank. Operations take and restore an integer number of tokens, either directly from the bank or from another reference with multiple tokens. An upgrade in read or write permission is equivalent to taking tokens from the bank and/or others, while a downgrade is equivalent to returning tokens to the bank. In terms of transformation rules, this approach corresponds to associating the + and − symbols in Figure 3 with Petri net arcs for taking tokens from and returning tokens to the common pile, respectively.

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:

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

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

To complete our model, in addition to the field *ap* in the basic module, we introduce two more fields, *tkr* and *tkw*, to represent the number of tokens of each type (reading and writing) that each reference $q = r_i^j$ holds. Then, the difference in number of tokens δ described below dictates the access permission transformations (upgrade, downgrade) from the currently owned number of tokens to the target number needed by ap':

$$\delta_r(q, ap') = N_r(ap') - s[q].tkr \in \{-(K+1), \dots, -1, 0, 1, \dots, K+1\}$$

$$\delta_w(q, ap') = N_w(ap') - s[q].tkw \in \{-(K+1), \dots, -1, 0, 1, \dots, K+1\}$$

A negative δ means downgrade, while a positive one means upgrade. The deficit (or surplus) is taken from (or given back to) the bank when available, otherwise the deficit is taken from other references in *post* state, without violating the requirements of the references in *pre* state.

We illustrate this new technique on the producer/consumer example presented in Section 2. The methods *isEmpty()* and *isFilled()* require non-exclusive rights (**Pure**), therefore the guard for starting *isEmpty()* checks whether the reference has the one read token necessary or it needs to "borrow" it from the bank or others. This results in two distinct types of transitions for *isEmpty()*:

- If $tkr_i^B + tkr_i^j \geq 1$, corresponding to $\delta_r(q, \textbf{Pure}) \geq 0$, then ${tkr_i^j}' = 1 \wedge {tkr_i^B}' = tkr_i^B + tkr_i^j - 1$
- If $tkr_i^B + tkr_i^j = 0 \wedge \exists h \neq j : pc_i^h = (\text{post}, \cdot) \wedge tkr_i^h \geq 1$, corresponding to $\delta_r(q, \textbf{Pure}) < 0$, then ${tkr_i^j}' = 1 \wedge {tkr_i^h}' = tkr_i^h - 1$ (one read token transferred from r_i^h to r_i^j)

Due to the existential quantifier in the guard of the second type of transition for *isEmpty()*, this results in K individual transitions of this type, one for each $h \neq j$.

The methods *consume()* and *produce()* require non-exclusive read rights and exclusive write rights. The exclusive rights are translated into requiring that no other reference is writing, $\forall h \neq j : tkw_i^h = 0$. All write tokens are then collected by r_i^j: ${tkw_i^j}' = K+1 \wedge {tkw_i^B}' = 0 \wedge \forall h \neq j : {tkw_i^h}' = 0$. The read rights requirements result in two subcases, similar to *isEmpty()*.

This model construction can be further optimized, by requiring that all methods return the tokens to the bank upon completion. This affects only the pre-to-post transitions: ${tkr_i^j}' = 0 \wedge {tkr_i^B}' = tkr_i^B + tkr_i^j$. Therefore, all available tokens at any time can be found only at the bank, which eliminates the need to check the other K references. With this observation, the K different transitions resulting from the existential quantifier are no longer needed. Besides brevity, this optimized model has the desired property of increased event locality (fewer dependencies between variables among different references), which dramatically improves the performance of the model checker.

5.1 The Model Generator

A C program is used to generate the `evmdd_smc` models in the framework just described. The input parameters to this program are c, the number of distinct objects, and K, the number of distinct references for each object. The program constructs a model that contains three sections: variable declarations, variable initializations, and the transition relation.

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 such variables. The domain $\mathcal{TS}_i = \{\bot\} \cup \{t_i^1, \ldots, t_i^{h_i}\}$ 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 $\mathcal{AP} = \{\bot, \mathbf{Unique}, \mathbf{Full}, \mathbf{Pure}, \mathbf{Immutable}, \mathbf{Share}\}$ is mapped to $[0, 5]$. For the variables referring to tokens, we use the domain $[0, K+1]$. For the $\{pre, post\}$ type we use $[0, 1]$.

Domain		concrete	abstract
Typestates	$\mathcal{TS}_i$	$\{\bot\} \cup \{t_i^1, \ldots, t_i^{h_i}\}$	$[0 \ldots h_i]$
Method ids	$\mathcal{M}_i$	$\{\bot\} \cup \{M_i^1, \ldots, M_i^{m_i}\}$	$[0 \ldots m_i]$
Access permissions	$\mathcal{AP}$	$\{\bot, \mathbf{Unique}, \mathbf{Full}, \mathbf{Pure}, \mathbf{Immutable}, \mathbf{Share}\}$	$[0 \ldots 5]$
Number of tokens		$[0 \ldots K+1]$	$[0 \ldots K+1]$
Program counter	$\mathcal{PC}$	$\{pre, post\}$	$[0 \ldots 1]$

b) *Variable declarations.*

For each object o_i, we declare two categories of variables. One category refers to the object proper 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.

Variable	name	type
Typestate	$state_i$	$[0 \ldots h_i]$
Read tokens in the bank	tkr_i^B	$[0 \ldots K+1]$
Write tokens in the bank	tkw_i^B	$[0 \ldots K+1]$
Program counter of r_i^j	pc_i^j	$[0 \ldots 1]$
Method executed by r_i^j	$method_i^j$	$[0 \ldots m_i]$
Access permission of r_i^j	ap_i^j	$[0 \ldots 5]$
Number of read tokens of r_i^j	tkr_i^j	$[0 \ldots K+1]$
Number of write tokens of r_i^j	tkw_i^j	$[0 \ldots K+1]$

c) *Variable initializations.*

For each object: $state_i = \bot \wedge tkr_i^B = K+1 \wedge tkw_i^B = K+1$.

For each reference: $0 \le j \le K$: $pc_i^j = \mathsf{post}$, $method_i^j = \bot$, $ap_i^j = \bot$, $tkr_i^j = tkw_i^j = 0$.

d) *Transition relation.*

Each transition has a guard expression (the enabling condition) and an update expression (the transformation performed by executing the transition). Unprimed variables refer to values before the update (the “from” state), while primed variables refer to values after the update (the "to" state). For each object o_i there is a set of transitions generated for each reference to the object r_i^j ($K+1$ sets). Similar to the translation algorithm in Section 4, guard and update expressions are generated from the **requires** and **ensures** clauses associated with the corresponding method in the specification. More precisely, the **requires** clause impacts the guard of starting a method, while the **ensures** clause impacts the update due to ending a method.

Therefore, we can identify four categories of transitions in our model, described in detail below.

1. Reference r_i^j starts a constructor.

   The guard only enforces that the object has not been created yet: $\bigwedge_{j=0}^{K}(ap_i^j = \bot)$.

   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: $pc_i^j{}' = \mathsf{pre} \wedge method_i^j{}' = \mathsf{constructor} \wedge ap_i^j{}' = \mathbf{Unique} \wedge tkr_i^B{}' = 0 \wedge tkw_i^B{}' = 0 \wedge tkr_i^j{}' = K+1 \wedge tkw_i^j{}' = K+1$.

2. Reference r_i^j starts a method m_i^k.

   The guard contains four conjuncts. The first requires r_i^j to exist (not undefined): $ap_i^j \neq \bot$. The second requires it to be in a post state (not executing something else): $pc_i^j = \mathsf{post}$. If the specification of m_i^k also requires that some object o_h (NB: must commonly but not necessarily the same o_i) to be in state t_h^x, the third conjunct enforces that: $state_h = t_h^x$.

   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: $pc_i^j{}' = \mathsf{pre} \wedge method_i^j{}' = k \wedge ap_i^j{}' = ap$. 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: $pc_i^j = \mathsf{pre} \wedge method_i^j = k$.

   The update expression reflects the change in the state of r_i^j: $pc_i^j{}' = \mathsf{post}$, 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 $pc_i^j{}' = \mathsf{post} \wedge method_i^j{}' = \bot \wedge ap_i^j{}' = \mathbf{Pure}$.

5.2 Properties of Interest

Within this framework, we can define several categories of generic properties that can be checked on any specification. They can be automatically generated and provide useful insight.

Sink states (Deadlock). 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 the CTL temporal logic [9], it can be expressed as simply as:

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

Once the state space is constructed, finding sink states is a very computationally inexpensive operation.

Typestate adjacency graph. Often, when laying out a specification, the designer knows in advance the expected control flow through the typestates of a class. This can be captured (and even depicted graphically) by means of the adjacency graph of the typestates. For instance, in a generic database application similar to the one presented in Section 5.3, the nominal flow of a task would be a linear graph traversing the typestates: $\perp \rightarrow \mathit{created} \rightarrow \mathit{ready} \rightarrow \mathit{completed} \rightarrow \mathit{destroyed}$.

In CTL, the adjacency relation can be written $\forall 1 \leq i \leq c, \forall t_1 \neq t_2 \in \mathcal{TS}_i$:

$$\mathit{adjacent}_i\,(t_1, t_2) : \mathit{state}_i = t_1 \wedge \mathrm{EX}(\mathit{state}_i = t_2)$$

Concurrency. Access permissions are abstractions used to represent how an object is used. These abstractions can be used for parallel execution of methods along with some other dependency information. The present Plural specification includes knowledge about access permissions but does not have explicit knowledge of other dependencies. Even with this partial knowledge only, we can infer certain facts about method execution order. For instance, we can find all the possible pairs of methods that can be executed in parallel and all the pairs of methods that can never be executed in parallel.

In CTL, $\forall 1 \leq i \leq c, 0 \leq j_1 \neq j_2 \leq K, \forall m_1 \neq m_2 \in \mathcal{M}_i$:

$$\mathit{concurrent}_i\,(m_1, m_2) : \mathrm{EF}\left(\mathit{pc}_i^{j_1} = (m_1, \mathit{pre}) \wedge \mathit{pc}_i^{j_2} = (m_2, \mathit{pre})\right)$$

Note that due to the symmetry in the model, it is sufficient to consider $j_1 = 0$ and $j_2 = 1$. An empty set of states satisfying property $\mathit{concurrent}_i\,(m_1, m_2)$ indicates that methods m_1 and m_2 can never be executed in parallel, while a non empty set means the opposite.

Correct use of access permission. A set of integrity checks can be performed to ensure that each access permission requirement attached to a method does not violate the intended semantics described in Section 2. This is theoretically enforced by the model construction presented in Section 5, therefore double-checking the construction would provide additional proof of correctness for the entire concept. Of the five access permissions, only violations of three have meaningful representations, as the remaining two (Pure and Share) do not impose restrictions on other coexisting references. Furthermore, the predicates for Full and Immutable are the same.

In CTL, $\forall 1 \leq i \leq c, \forall m \in \mathcal{M}_i, \forall 0 \leq j_1 \neq j_2 \leq K$:

$$\mathit{unique\_access}(m) : \mathrm{AG}\neg \Big(pc_i^{j_1} = (m, \mathit{pre}) \wedge ap_i^{j_2} \neq \bot \Big)$$

$$\mathit{full\_or\_imm\_access}(m) : \mathrm{AG}\neg \Big(pc_i^{j_1} = (m, \mathit{pre}) \wedge pc_i^{j_2} = (\cdot, \mathit{pre}) \wedge \mathit{tkw}_i^{j_2} > 0 \Big)$$

5.3 Application

We present a case study on a Plural specification taken from a real database application, that has been previously used in Plural experiments. We first present an actual flawed attempt to write the specification and show how model checking can help expose and then correct the errors.

```
@Refine({
  @States(dim = "A", value={"Created", "Ready", "Comlpleted", "Destroyed"}),
  @States(dim = "B", value={"U_Data", "P_Data", "F_Data"}),
})
@ClassStates({
  @State(name = "Created",   inv ="data!=null"),
  @State(name = "Ready",     inv ="data!=null"),
  @State(name = "Completed", inv ="data!=null"),
  @State(name = "Destroyed", inv ="data==null"),

  @State(name = "U_Data", inv="Unique(data)"),
  @State(name = "F_Data", inv="Full(data)"),
  @State(name = "P_Data", inv="Pure(data)")
})

public class mttsTask {

  private MttsTaskDataX data;

  @Perm(ensures= "Unique(this) in Created")
  @Unique(ensures="U_Data")
  mttsTask()
  { data=new MttsTaskDataX(); }

  @Fulls({
  @Full(requires="Created",ensures="Ready"),
  @Full(requires="F_Data", ensures="F_Data")})
  public void setData(MttsTaskDataX data)
  { ... }

  @Pure(requires="P_Data", ensures=‘P_Data")
  public MttsTaskDataX getData()
  { return data; }

  @Full (requires="Ready",ensures= "Completed")
  public void execute()
  { ... }

  @Full(requires="Complete",ensures= "Destroyed")
  public void delete()
  { data=null; }
}
```

Figure 7. Specification of `mttsTask.java` with errors.

The class `mttsTask` models a generic processing task in the database system. The internal information about the task is stored in a private member `data` of type `mttsTaskDataX`. The constructor of class `mttsTask` creates a **Unique** object that is initially in state `Created`. The method `setData()` requires **this** to have **Full** permission on its referenced object, which should be in state `Created`. 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 Completed. The method delete() sets data to null and moves the task to the Destroyed typestate.

```
@ClassStates({
  @State(name = "Created",   inv="data!=null"),
  @State(name = "Ready",     inv="data!=null"),
  @State(name = "Completed", inv="data!=null"),
  @State(name = "Destroyed", inv="data==null"),
})

public class mttsTask {
  private MttsTaskDataX data;

  @Perm (ensures= "unique(this) in Created")
  mttsTask()
    { data=new MttsTaskDataX(); }

  @Full(requires="Created", ensures= "Ready")
  @Perm(requires="#0!=null")
  public void setData(MttsTaskDataX data)
    { this.data=data; }

  @Pure(requires="Ready", esnures="Ready")
  public MttsTaskDataX getData()
    { return data; }

  @Full(requires="Ready",ensures= "Completed")
  public void execute() { }

  @Full(requires="Completed", ensures= "Destroyed")
  public void delete()
    { data=null; }
}
```

Figure 8. Corrected specification of mttsTask.java

Note that the typestate hierarchy in this example has two *dimensions* (labeled A and B). Dimensions were not discussed in previous sections. They define a Cartesian product of possible states an object can have. Dimensions are modeled very straightforwardly by using two distinct variables (stateA and stateB), one for each dimension. The state-space for the typestates is simply the cross product of the two subdomains. In this example, the specification of the second dimension is trying to mimic the access permissions on the field data. The notation ‘‘#i’’ is used to refer to the i^{th} argument of a method. In this case, the 0^{th} argument of setData() is of type mttsTaskDataX.

Finally, one other feature not covered so far are the invariants. In their most general form, they cannot be captured by our automated approach. However, in this instance, the invariants are simple equality checks on data. This is captured by a single boolean variable that represents whether data is null or not.

As mentioned before, the specification in Figure 7 contains errors. However, the Plural tool does not generate any warning or error message. The first error is a simple syntax error in method delete(). The typestate Completed is misspelled as Complete. With this incorrect specification, the method delete() cannot be called after the method execute().

The second is a semantic error. After the constructor call, the method setData() cannot be invoked. Similarly, getData() cannot be called after setData(). The constructor takes the object to state U_Data, while setData() requires state F_Data. Using the model checker, the reachability analysis is able to expose the error, by signaling that the adjacency graph on typestates is disconnected.

The root cause of the semantic error is not just the incomplete adjacency matrix on typestates in the first dimension. The attempt to follow access permissions by manipulating typestates on the second dimension is bound to fail, as access permissions provide a much more expressive environment, by means of transformations (upgrades/downgrades), whereas the typestate predicates can operate only with equality. For this reason, in the correct implementation the second dimension is dropped. Figures 8 and 9 present a working version of this specification.

The specification of `mttsTaskDataX.java` contains no typestates (which is in fact encoded in practice with a single default value `alive`) and seven methods (including the constructor).

```
public class MttsTaskDataX {
  private int task_ID;
  private int task_priority;
  private Set task_dependencies;

  @Perm(ensures="unique(this)")
  MttsTaskDataX() { ... }

  @Full
  void setID(int id)
    { task_ID=id; }

  @Full
  void setPriority(int priority)
    { task_priority=priority; }

  @Full
  void setDependencies(Set dependencies)
    { task_dependencies=dependencies; }

  @Pure
  int getID(int id) { ... }

  @Full
  int getPriority() { ... }

  @Full
  Set getDependencies() { ... }

}
```

Figure 9. Specification of `mttsTaskDataX.java`

A listing of the model generator for this application (with two classes) is given in Appendix C. This more advanced version paves the way for full automation with templates for starting/ending generic methods and dealing with parameters.

For reference, we have run scalability experiments on the complete `mttsTask` model. We list the size of the state space and the total verification time for each K. Note that in practice it is sufficient to use $K < 6$. The table shows that the model checker can easily handle extremely large state spaces: more than 10^{37} states in less than a minute.

6 Future Work

There are several extensions (and refinements) possible to our basic approach. First and foremost, we need to incorporate the additional constraints that class relationships (inheritance, containment) impose on state transitions rules; Access permission of method arguments also impact the analysis but is not currently taken into account. We consider that an incremental

K	# states	time (sec.)	K	# states	time (sec.)
1	1.7×10^{4}	0.03	11	1.3×10^{22}	3.80
2	1.5×10^{6}	0.12	12	6.6×10^{23}	4.62
3	1.2×10^{8}	0.27	13	3.1×10^{25}	5.40
4	8.4×10^{9}	0.48	14	1.4×10^{27}	6.70
5	5.4×10^{11}	0.73	15	6.5×10^{28}	7.52
6	3.3×10^{13}	1.07	16	2.9×10^{30}	9.26
7	1.8×10^{15}	1.49	17	1.2×10^{32}	9.80
8	1.0×10^{17}	1.98	18	5.5×10^{33}	11.37
9	5.4×10^{18}	2.49	19	2.3×10^{35}	13.19
10	2.7×10^{20}	3.18	20	1.0×10^{37}	14.28

Figure 10. Experimental results: scalability for the `mttsTask` model.

refinement of the overall abstraction is possible, by augmenting the models with member variables, explicit typestate invariants, etc. One instance is to avoid explicitly encoding the access permissions into the states for all possible combinations, but instead use a deductive method (theorem proving, 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 employs discrete-state systems, and using more powerful, deductive techniques: SAT/SMT solvers or automated theorem provers. We plan to implement our approach as part of the Plural tool.

Aknowledgments. We would like to thank Ijaz Ahmad for the many constructive comments that helped improve the text and for his contributions to the model generator code.

References

1. Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 1015–1022, New York, NY, USA, 2009. ACM.

2. Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN*, pages 103–122, 2001.

3. K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *OOPSLA*. ACM, 2007.

4. Kevin Bierhoff. Iterator specification with typestates. In *Proceedings of the 2006 conference on Specification and verification of component-based systems*, SAVCBS '06, pages 79–82, New York, NY, USA, 2006. ACM.

5. Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical API protocol checking with access permissions, 2009.

6. N. Beckman K. Bierhoff and J. Aldrich. Verifying correct usage of atomic blocks and typestate. In *OOPSLA*, 2008.

7. John Boyland. Checking interference with fractional permissions. In *Static Analysis: 10th International Symposium*, 2003.

8. N. Cataño and T. Wahls. Executing JML specifications of java card applications: A case study. In E. Wong, C. Sung, and S. Ghosh, editors, *24th ACM Symposium on Applied Computing, Software Engineering Track (SAC-SE)*, pages 404–408, Waikiki Beach, Honolulu, Hawaii, March 8-12 2009.

9. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2):244–263, April 1986.

10. Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, pages 59–69, 2001.

11. Andy Galloway, Gerald Lüttgen, Jan Tobias Mühlberg, and Radu Siminiceanu. Model-checking the linux virtual file system. In *Verification, Model Checking, and Abstract Interpretation, 10th International Conference (VMCAI 2009), Savannah*, volume 5403 of *Lecture Notes in Computer Science*, pages 74–88. Springer, 2009.

12. Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

13. Ben Krause and Tim Wahls. jmle: A tool for executing JML specifications via constraint programming. In L. Brim, editor, *Formal Methods for Industrial Critical Systems (FMICS '06)*, volume 4346 of *Lecture Notes in Computer Science*, pages 293 – 296. Springer-Verlag, August 2006.

14. G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT (Software Engineering Symposium)*, 31(3):1–38, 2006.

15. The Plural Tool. `http://code.google.com/p/pluralism/`.

16. Pierre Roux and Radu Siminiceanu. Model checking with edge-valued decision diagrams. In César Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215*, pages 222–226, Langley Research Center, Hampton VA 23681-2199, USA, April 2010. NASA.

17. Radu Siminiceanu, Rick W. Butler, and César A. Muñoz. Experimental evaluation of a planning language suitable for formal verification. In *Model Checking and Artificial Intelligence, 5th International Workshop (MoChArt 2008), Patras, Greece*, volume 5348 of *Lecture Notes in Computer Science*, pages 132–146. Springer, 2008.

18. Radu I. Siminiceanu. Model checking abstract PLEXIL programs with SMART. Technical Report NASA/CR-2007-214542, NASA Langley Research Center, Hampton, VA, 2007.

19. S. Stork, P. Marques, and J. Aldrich. Concurrency by Default: Using Permissions to Express Dataflow in Stateful Programs. In *Onward! Conference*, October 2009.

20. R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions of Software Engineering*, 12(1):157–171, 1986.

Appendix A

C program to generate the producer/consumer model for `evmdd_smc`

Note that, in this example, we have only one class definition, therefore, for simplicity, we can ignore the subscript i that identifies references to object i. Only superscript j, ranging from 0 to K, will appear.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define UNDEF           0
#define UNIQUE          1
#define FULL            2
#define SHARED          3
#define PURE            4
#define IMMUTABLE       5
#define NUM_AP          5

#define EMPTY           1
#define PARTIAL         2
#define FILLED          3
#define NUM_STATES      3

#define CONSTRUCTOR     1
#define IS_EMPTY        2
#define IS_FILLED       3
#define PRODUCE         4
#define CONSUME         5
#define NUM_METHODS     5

#define PRE             0
#define POST            1

int K;

void Init(){
   printf("Declarations\n");

   printf("  state    [0, %d]\n", NUM_STATES);
   printf("  tkrB     [0, %d]\n", K+1);
   printf("  tkwB     [0, %d]\n\n", K+1);
   for (int i=0; i<=K; i++) {
     printf("  pc_%d      [0, 1]\n", i);
     printf("  method_%d  [0, %d]\n", i, NUM_METHODS);
     printf("  ap_%d      [0, %d]\n", i, NUM_AP);
     printf("  tkr_%d     [0, %d]\n", i, K+1);
     printf("  tkw_%d     [0, %d]\n\n", i, K+1);
   }

   printf("Initial states\n");
   printf("  state    = %d\n", UNDEF);
   printf("  tkrB     = %d\n", K+1);
   printf("  tkwB     = %d\n\n", K+1);
   for (int i=0; i<=K; i++) {
     printf("  pc_%d     = %d\n", i, POST);
     printf("  method_%d = %d\n", i, UNDEF);
     printf("  ap_%d     = %d\n", i, UNDEF);
     printf("  tkr_%d    = 0\n", i);
     printf("  tkw_%d    = 0\n\n", i);
   }
}

void Trans(){
   printf("Transitions\n");
   for (int i=0; i<=K; i++) {
```

```
// start constructor, unique access
printf(" start_constructor_%d:\n", i);
printf("  ap_%d = %d", i, UNDEF);
for (int j=0; j<=K; j++) if (i!=j) printf(" /\\ ap_%d = %d", j, UNDEF);
printf("  ->");
printf("  pc_%d\’ = %d /\\ method_%d\’ = %d /\\ ap_%d\’ = %d", i, PRE, i,
     CONSTRUCTOR, i, UNIQUE);
printf("  /\\ tkrB\’ = 0 /\\ tkwB\’ = 0 /\\ tkr_%d\’ = %d /\\ tkw_%d\’ = %d\n", i, K
     +1, i, K+1);

// start isEmpty(), 1 read token available
printf(" start_is_empty_%d:\n", i);
printf("  pc_%d = %d /\\ ap_%d != %d /\\ tkrB > 0", i, POST, i, UNDEF);
printf("  ->");
printf("  pc_%d\’ = %d /\\ method_%d\’ = %d /\\ ap_%d\’ = %d", i, PRE, i, IS_EMPTY,
     i, PURE);
printf("  /\\ tkr_%d\’ = 1 /\\ tkrB\’ = tkrB - 1\n", i);

// start isFilled(), 1 read token available
printf(" start_is_filled_%d:\n", i);
printf("  pc_%d = %d /\\ ap_%d != %d /\\ tkrB > 0", i, POST, i, UNDEF);
printf("  ->");
printf("  pc_%d\’ = %d /\\ method_%d\’ = %d /\\ ap_%d\’ = %d", i, PRE, i, IS_FILLED,
      i, PURE);
printf("  /\\ tkr_%d\’ = 1 /\\ tkrB\’ = tkrB - 1\n", i);

// start produce(), enough tokens exist
printf(" start_produce_%d:\n", i);
printf("  pc_%d = %d /\\ ap_%d != %d /\\ (state = %d \\/ state = %d)", i, POST, i,
     UNDEF, EMPTY, PARTIAL);
printf("  /\\ tkrB > 0 /\\ tkwB = %d", K+1);
printf("  ->");
printf("  pc_%d\’ = %d /\\ method_%d\’ = %d /\\ ap_%d\’ = %d", i, PRE, i, PRODUCE, i
     , FULL);
printf("  /\\ tkwB\’ = 0 /\\ tkrB\’ = tkrB - 1 /\\ tkr_%d\’ = 1 /\\ tkw_%d\’ = %d\n"
     , i, i, K+1);

// start consume(), enough tokens exist
printf(" start_consume_%d:\n", i);
printf("  pc_%d = %d /\\ ap_%d != %d /\\ (state = %d \\/ state = %d)", i, POST, i,
     UNDEF, FILLED, PARTIAL);
printf("  /\\ tkrB > 0 /\\ tkwB = %d", K+1);
printf("  ->");
printf("  pc_%d\’ = %d /\\ method_%d\’ = %d /\\ ap_%d\’ = %d", i, PRE, i, CONSUME, i
     , FULL);
printf("  /\\ tkwB\’ = 0 /\\ tkrB\’ = tkrB - 1 /\\ tkr_%d\’ = 1 /\\ tkw_%d\’ = %d\n"
     , i, i, K+1);

// end methods
printf(" end_constructor_%d:\n", i);
printf("  pc_%d = %d /\\ method_%d = %d  ->  pc_%d\’ = %d /\\ state\’ = %d", i, PRE,
      i, CONSTRUCTOR, i, POST, EMPTY);
printf("  /\\ tkrB’ = %d /\\ tkwB\’ = %d /\\ tkr_%d\’ = 0 /\\ tkw_%d\’ = 0\n", K+1,
     K+1, i, i);
printf(" end_is_empty_%d:\n", i);
printf("  pc_%d = %d /\\ method_%d = %d  ->  pc_%d\’ = %d", i, PRE, i, IS_EMPTY, i,
     POST);
printf("  /\\ tkrB’ = tkrB + 1 /\\ tkr_%d\’ = 0\n", i);
printf(" end_is_filled_%d:\n", i);
printf("  pc_%d = %d /\\ method_%d = %d  ->  pc_%d\’ = %d", i, PRE, i, IS_FILLED, i,
      POST);
printf("  /\\ tkrB’ = tkrB + 1 /\\ tkr_%d\’ = 0\n", i);
printf(" end_produce_1st_%d:\n", i);
printf("  pc_%d = %d /\\ method_%d = %d /\\ state = %d  ->  pc_%d\’ = %d /\\ state\’
      = %d", i, PRE, i, PRODUCE, EMPTY, i, POST, PARTIAL);
printf("  /\\ tkrB’ = tkrB + 1 /\\ tkwB\’ = %d /\\ tkr_%d\’ = 0 /\\ tkw_%d\’ = 0\n",
      K+1, i, i);
printf(" end_produce_%d:\n", i);
printf("  pc_%d = %d /\\ method_%d = %d /\\ state = %d  ->  pc_%d\’ = %d /\\ (state
     \’ = %d \\/ state\’ = %d)", i, PRE, i, PRODUCE, PARTIAL, i, POST, PARTIAL,
     FILLED);
printf("  /\\ tkrB’ = tkrB + 1 /\\ tkwB\’ = %d /\\ tkr_%d\’ = 0 /\\ tkw_%d\’ = 0\n",
      K+1, i, i);
printf(" end_consume_full_%d:\n", i);
```

```
     printf("  pc_%d = %d /\\ method_%d = %d /\\ state = %d  ->  pc_%d\' = %d /\\ state\'
          = %d", i, PRE, i, CONSUME, FILLED, i, POST, PARTIAL);
     printf("  /\\ tkrB' = tkrB + 1 /\\ tkwB\' = %d /\\ tkr_%d\' = 0 /\\ tkw_%d\' = 0\n",
          K+1, i, i);
     printf(" end_consume_%d:\n", i);
     printf("  pc_%d = %d /\\ method_%d = %d /\\ state = %d  ->  pc_%d\' = %d /\\ (state
         \' = %d \\/ state\' = %d)", i, PRE, i, CONSUME, PARTIAL, i, POST, PARTIAL,
         EMPTY);
     printf("  /\\ tkrB' = tkrB + 1 /\\ tkwB\' = %d /\\ tkr_%d\' = 0 /\\ tkw_%d\' = 0\n",
          K+1, i, i);

     // create alias, read tokens exist
     printf(" create_alias_%d:\n", i);
     printf("  ap_%d = %d /\\ tkrB > 0 /\\ state != %d", i, UNDEF, UNDEF);
     printf("  ->");
     printf("  pc_%d\' = %d /\\ method_%d\' = %d /\\ ap_%d\' = %d\n", i, POST, i, UNDEF,
         i, PURE);

   }

}

void Spec() {
  printf("Properties\n");
  printf("  !EX(true)\n");
}

int main(int argc, char *argv[])
{
  K = 5;
  for (int i = 1; i < argc; ++i) {
    if (strcmp(argv[i], "-k")==0 && i<argc-1) {
      K = atoi(argv[i+1]);
      if (K<0) {
        fprintf(stderr, " ERROR: invalid number of references, %d.\n", K);
        fprintf(stderr, "        need at least 1 reference.\n");
        return 2;
      }
    }
  }

  Init();
  Trans();
  Spec();
  return 0;
}
```

Appendix B

An example generated evmdd_smc model

Below is an instance of a generated producer/consumer model, for parameter $K = 2$.

```
Declarations
  state     [0, 3]
  tkrB      [0, 3]
  tkwB      [0, 3]

  pc_0      [0, 1]
  method_0  [0, 5]
  ap_0      [0, 5]
  tkr_0     [0, 3]
  tkw_0     [0, 3]

  pc_1      [0, 1]
  method_1  [0, 5]
  ap_1      [0, 5]
  tkr_1     [0, 3]
  tkw_1     [0, 3]

  pc_2      [0, 1]
  method_2  [0, 5]
  ap_2      [0, 5]
  tkr_2     [0, 3]
  tkw_2     [0, 3]

Initial states
  state    = 0
  tkrB     = 3
  tkwB     = 3

  pc_0     = 1
  method_0 = 0
  ap_0     = 0
  tkr_0    = 0
  tkw_0    = 0

  pc_1     = 1
  method_1 = 0
  ap_1     = 0
  tkr_1    = 0
  tkw_1    = 0

  pc_2     = 1
  method_2 = 0
  ap_2     = 0
  tkr_2    = 0
  tkw_2    = 0

Transitions
 start_constructor_0:
   ap_0 = 0 /\ ap_1 = 0 /\ ap_2 = 0  ->
   pc_0’ = 0 /\ method_0’ = 1 /\ ap_0’ = 1  /\ tkrB’ = 0 /\ tkwB’ = 0 /\ tkr_0’ = 3 /\
        tkw_0’ = 3
 start_is_empty_0:
   pc_0 = 1 /\ ap_0 != 0 /\ tkrB > 0  ->
   pc_0’ = 0 /\ method_0’ = 2 /\ ap_0’ = 4  /\ tkr_0’ = 1 /\ tkrB’ = tkrB - 1
 start_is_filled_0:
   pc_0 = 1 /\ ap_0 != 0 /\ tkrB > 0  ->
   pc_0’ = 0 /\ method_0’ = 3 /\ ap_0’ = 4  /\ tkr_0’ = 1 /\ tkrB’ = tkrB - 1
 start_produce_0:
   pc_0 = 1 /\ ap_0 != 0 /\ (state = 1 \/ state = 2)  /\ tkrB > 0 /\ tkwB = 3  ->
   pc_0’ = 0 /\ method_0’ = 4 /\ ap_0’ = 2  /\ tkwB’ = 0 /\ tkrB’ = tkrB - 1 /\ tkr_0’ =
        1 /\ tkw_0’ = 3
 start_consume_0:
   pc_0 = 1 /\ ap_0 != 0 /\ (state = 3 \/ state = 2)  /\ tkrB > 0 /\ tkwB = 3  ->
   pc_0’ = 0 /\ method_0’ = 5 /\ ap_0’ = 2  /\ tkwB’ = 0 /\ tkrB’ = tkrB - 1 /\ tkr_0’ =
        1 /\ tkw_0’ = 3
```

```
end_constructor_0:
  pc_0 = 0 /\ method_0 = 1  ->
  pc_0’ = 1 /\ state’ = 1  /\ tkrB’ = 3 /\ tkwB’ = 3 /\ tkr_0’ = 0 /\ tkw_0’ = 0
end_is_empty_0:
  pc_0 = 0 /\ method_0 = 2  ->
  pc_0’ = 1  /\ tkrB’ = tkrB + 1 /\ tkr_0’ = 0
end_is_filled_0:
  pc_0 = 0 /\ method_0 = 3  ->
  pc_0’ = 1  /\ tkrB’ = tkrB + 1 /\ tkr_0’ = 0
end_produce_1st_0:
  pc_0 = 0 /\ method_0 = 4 /\ state = 1  ->
  pc_0’ = 1 /\ state’ = 2  /\ tkrB’ = tkrB + 1 /\ tkwB’ = 3 /\ tkr_0’ = 0 /\ tkw_0’ = 0
end_produce_0:
  pc_0 = 0 /\ method_0 = 4 /\ state = 2  ->
  pc_0’ = 1 /\ (state’ = 2 \/ state’ = 3)  /\ tkrB’ = tkrB + 1 /\ tkwB’ = 3 /\ tkr_0’ =
      0 /\ tkw_0’ = 0
end_consume_full_0:
  pc_0 = 0 /\ method_0 = 5 /\ state = 3  ->
  pc_0’ = 1 /\ state’ = 2  /\ tkrB’ = tkrB + 1 /\ tkwB’ = 3 /\ tkr_0’ = 0 /\ tkw_0’ = 0
end_consume_0:
  pc_0 = 0 /\ method_0 = 5 /\ state = 2  ->
  pc_0’ = 1 /\ (state’ = 2 \/ state’ = 1)  /\ tkrB’ = tkrB + 1 /\ tkwB’ = 3 /\ tkr_0’ =
      0 /\ tkw_0’ = 0
create_alias_0:
  ap_0 = 0 /\ tkrB > 0 /\ state != 0  ->
  pc_0’ = 1 /\ method_0’ = 0 /\ ap_0’ = 4
start_constructor_1:
  ap_1 = 0 /\ ap_0 = 0 /\ ap_2 = 0  ->
  pc_1’ = 0 /\ method_1’ = 1 /\ ap_1’ = 1  /\ tkrB’ = 0 /\ tkwB’ = 0 /\ tkr_1’ = 3 /\
      tkw_1’ = 3
start_is_empty_1:
  pc_1 = 1 /\ ap_1 != 0 /\ tkrB > 0  ->
  pc_1’ = 0 /\ method_1’ = 2 /\ ap_1’ = 4  /\ tkr_1’ = 1 /\ tkrB’ = tkrB - 1
start_is_filled_1:
  pc_1 = 1 /\ ap_1 != 0 /\ tkrB > 0  ->
  pc_1’ = 0 /\ method_1’ = 3 /\ ap_1’ = 4  /\ tkr_1’ = 1 /\ tkrB’ = tkrB - 1
start_produce_1:
  pc_1 = 1 /\ ap_1 != 0 /\ (state = 1 \/ state = 2)  /\ tkrB > 0 /\ tkwB = 3  ->
  pc_1’ = 0 /\ method_1’ = 4 /\ ap_1’ = 2  /\ tkwB’ = 0 /\ tkrB’ = tkrB - 1 /\ tkr_1’ =
      1 /\ tkw_1’ = 3
start_consume_1:
  pc_1 = 1 /\ ap_1 != 0 /\ (state = 3 \/ state = 2)  /\ tkrB > 0 /\ tkwB = 3  ->
  pc_1’ = 0 /\ method_1’ = 5 /\ ap_1’ = 2  /\ tkwB’ = 0 /\ tkrB’ = tkrB - 1 /\ tkr_1’ =
      1 /\ tkw_1’ = 3
end_constructor_1:
  pc_1 = 0 /\ method_1 = 1  ->
  pc_1’ = 1 /\ state’ = 1  /\ tkrB’ = 3 /\ tkwB’ = 3 /\ tkr_1’ = 0 /\ tkw_1’ = 0
end_is_empty_1:
  pc_1 = 0 /\ method_1 = 2  ->
  pc_1’ = 1  /\ tkrB’ = tkrB + 1 /\ tkr_1’ = 0
end_is_filled_1:
  pc_1 = 0 /\ method_1 = 3  ->
  pc_1’ = 1  /\ tkrB’ = tkrB + 1 /\ tkr_1’ = 0
end_produce_1st_1:
  pc_1 = 0 /\ method_1 = 4 /\ state = 1  ->
  pc_1’ = 1 /\ state’ = 2  /\ tkrB’ = tkrB + 1 /\ tkwB’ = 3 /\ tkr_1’ = 0 /\ tkw_1’ = 0
end_produce_1:
  pc_1 = 0 /\ method_1 = 4 /\ state = 2  ->
  pc_1’ = 1 /\ (state’ = 2 \/ state’ = 3)  /\ tkrB’ = tkrB + 1 /\ tkwB’ = 3 /\ tkr_1’ =
      0 /\ tkw_1’ = 0
end_consume_full_1:
  pc_1 = 0 /\ method_1 = 5 /\ state = 3  ->
  pc_1’ = 1 /\ state’ = 2  /\ tkrB’ = tkrB + 1 /\ tkwB’ = 3 /\ tkr_1’ = 0 /\ tkw_1’ = 0
end_consume_1:
  pc_1 = 0 /\ method_1 = 5 /\ state = 2  ->
  pc_1’ = 1 /\ (state’ = 2 \/ state’ = 1)  /\ tkrB’ = tkrB + 1 /\ tkwB’ = 3 /\ tkr_1’ =
      0 /\ tkw_1’ = 0
create_alias_1:
  ap_1 = 0 /\ tkrB > 0 /\ state != 0  ->
  pc_1’ = 1 /\ method_1’ = 0 /\ ap_1’ = 4
start_constructor_2:
  ap_2 = 0 /\ ap_0 = 0 /\ ap_1 = 0  ->
  pc_2’ = 0 /\ method_2’ = 1 /\ ap_2’ = 1  /\ tkrB’ = 0 /\ tkwB’ = 0 /\ tkr_2’ = 3 /\
      tkw_2’ = 3
start_is_empty_2:
    pc_2 = 1 /\ ap_2 != 0 /\ tkrB > 0  ->
    pc_2' = 0 /\ method_2' = 2 /\ ap_2' = 4  /\ tkr_2' = 1 /\ tkrB' = tkrB - 1
  start_is_filled_2:
    pc_2 = 1 /\ ap_2 != 0 /\ tkrB > 0  ->
    pc_2' = 0 /\ method_2' = 3 /\ ap_2' = 4  /\ tkr_2' = 1 /\ tkrB' = tkrB - 1
  start_produce_2:
    pc_2 = 1 /\ ap_2 != 0 /\ (state = 1 \/ state = 2)  /\ tkrB > 0 /\ tkwB = 3  ->
    pc_2' = 0 /\ method_2' = 4 /\ ap_2' = 2  /\ tkwB' = 0 /\ tkrB' = tkrB - 1 /\ tkr_2' =
         1 /\ tkw_2' = 3
  start_consume_2:
    pc_2 = 1 /\ ap_2 != 0 /\ (state = 3 \/ state = 2)  /\ tkrB > 0 /\ tkwB = 3  ->
    pc_2' = 0 /\ method_2' = 5 /\ ap_2' = 2  /\ tkwB' = 0 /\ tkrB' = tkrB - 1 /\ tkr_2' =
         1 /\ tkw_2' = 3
  end_constructor_2:
    pc_2 = 0 /\ method_2 = 1  ->
    pc_2' = 1 /\ state' = 1  /\ tkrB' = 3 /\ tkwB' = 3 /\ tkr_2' = 0 /\ tkw_2' = 0
  end_is_empty_2:
    pc_2 = 0 /\ method_2 = 2  ->
    pc_2' = 1  /\ tkrB' = tkrB + 1 /\ tkr_2' = 0
  end_is_filled_2:
    pc_2 = 0 /\ method_2 = 3  ->
    pc_2' = 1  /\ tkrB' = tkrB + 1 /\ tkr_2' = 0
  end_produce_1st_2:
    pc_2 = 0 /\ method_2 = 4 /\ state = 1  ->
    pc_2' = 1 /\ state' = 2  /\ tkrB' = tkrB + 1 /\ tkwB' = 3 /\ tkr_2' = 0 /\ tkw_2' = 0
  end_produce_2:
    pc_2 = 0 /\ method_2 = 4 /\ state = 2  ->
    pc_2' = 1 /\ (state' = 2 \/ state' = 3)  /\ tkrB' = tkrB + 1 /\ tkwB' = 3 /\ tkr_2' =
         0 /\ tkw_2' = 0
  end_consume_full_2:
    pc_2 = 0 /\ method_2 = 5 /\ state = 3  ->
    pc_2' = 1 /\ state' = 2  /\ tkrB' = tkrB + 1 /\ tkwB' = 3 /\ tkr_2' = 0 /\ tkw_2' = 0
  end_consume_2:
    pc_2 = 0 /\ method_2 = 5 /\ state = 2  ->
    pc_2' = 1 /\ (state' = 2 \/ state' = 1)  /\ tkrB' = tkrB + 1 /\ tkwB' = 3 /\ tkr_2' =
         0 /\ tkw_2' = 0
  create_alias_2:
    ap_2 = 0 /\ tkrB > 0 /\ state != 0  ->
    pc_2' = 1 /\ method_2' = 0 /\ ap_2' = 4
Properties
  !EX(true)
```

Appendix C

The generator for the mttsTask model

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int K;

#define UNDEF     0
#define UNIQUE    1
#define FULL      2
#define SHARED    3
#define PURE      4
#define IMMUTABLE 5
#define NUM_AP    5

#define PRE     0
#define POST    1

//MttsTask.java States
#define CREATED   1
#define READY     2
#define COMPLETE  3
#define DESTROYED 4
#define NUM_STATES_MttsTask 4

//MttsTask.java methods
#define MttsTask  1
#define setData   2
#define getData   3
#define execute   4
#define DELETE    5
#define NUM_METHODS_MttsTask  5

//MttsTaskDataX.java states
#define ALIVE       1
#define NUM_STATES_MttsTaskDataX  1

//MttsTaskDataX.java methods
#define MttsTaskDataX   1
#define setID           2
#define setPriority     3
#define setDependencies 4
#define getID           5
#define getPriority     6
#define getDependencies 7
#define NUM_METHODS_MttsTaskDataX 7

#define MttsTask_CLASS      0
#define MttsTaskDataX_CLASS 1
#define NUM_CLASSES         2
const int NUM_STATES[NUM_CLASSES] =
  {NUM_STATES_MttsTask, NUM_STATES_MttsTaskDataX};
const int NUM_METHODS[NUM_CLASSES] =
  {NUM_METHODS_MttsTask, NUM_METHODS_MttsTaskDataX};

void Init() {
  printf("Declarations\n");
  for (int i=0; i<NUM_CLASSES; i++) {
    printf("  state_%d   [0, %d]\n", i, NUM_STATES[i]);
    printf("  tkrB_%d    [0, %d]\n", i, K+1);
    printf("  tkwB_%d    [0, %d]\n\n", i, K+1);
    for (int j=0; j<=K; j++) {
      printf("  pc_%d_%d      [0, 1]\n", i, j);
      printf("  method_%d_%d  [0, %d]\n", i, j, NUM_METHODS[i]);
      printf("  ap_%d_%d      [0, %d]\n", i, j, NUM_AP);
      printf("  tkr_%d_%d     [0, %d]\n", i, j, K+1);
      printf("  tkw_%d_%d     [0, %d]\n\n", i, j, K+1);
    }
  }
  printf("Initial states\n");
  for (int i=0; i<NUM_CLASSES; i++) {
    printf("  state_%d   = %d\n", i, UNDEF);
    printf("  tkrB_%d     = %d\n", i, K+1);
    printf("  tkwB_%d     = %d\n\n", i, K+1);
    for (int j=0; j<=K; j++) {
      printf("  pc_%d_%d     = %d\n", i, j, POST);
      printf("  method_%d_%d = %d\n", i, j, UNDEF);
      printf("  ap_%d_%d     = %d\n", i, j, UNDEF);
      printf("  tkr_%d_%d    = 0\n", i, j);
      printf("  tkw_%d_%d    = 0\n\n", i, j);
    }
  }
}

void start_constructor(int classID,const char* name, int id, int state, int j )
{
 printf(" start_%s_%d:\t", name,j);
  for (int k=0; k<=K; k++){
      if (k!=K)  printf(" ap_%d_%d = %d /\\",classID, k, UNDEF);
      else    printf(" ap_%d_%d = %d", classID,k, UNDEF);
  }
  printf("  ->");
  printf("  pc_%d_%d\' = %d /\\ method_%d_%d\' = %d",classID, j, PRE, classID,j, id);
  printf("  /\\ ap_%d_%d\' = %d /\\ tkrB_%d\' = 0 /\\ tkwB_%d\' = 0 /\\ tkr_%d_%d\' = %d
      /\\ tkw_%d_%d\' = %d\n",classID, j, UNIQUE,classID,classID, classID,j, K+1,classID
      , j, K+1);
}

void end_constructor(int classID,const char* name, int id, int state, int j )
{
  printf(" end_%s_%d:\t",name, j);
  printf("  pc_%d_%d = %d  /\\ method_%d_%d = %d ",classID, j, PRE, classID,j,id);
  printf("  -> ");
  printf(" pc_%d_%d\' = %d", classID,j, POST);
  if (state!=-1)  printf(" /\\ state_%d\' = %d ",classID,state);
  printf("  /\\ tkrB_%d' = %d /\\ tkwB_%d\' = %d /\\ tkr_%d_%d\' = 0 /\\ tkw_%d_%d\' = 0\
      n",classID, K+1,classID, K+1,classID, j, classID,j);
}

void start_ap_state(int classID, int ap, int state, int j)
{
  if (ap==FULL)
    printf("  /\\ ap_%d_%d != %d /\\ tkrB_%d > 0 /\\ tkwB_%d = %d", classID,j, UNDEF,
         classID,classID, K+1);
  else if (ap==PURE)
    printf("  /\\ ap_%d_%d != %d /\\ tkrB_%d > 0", classID,j, UNDEF,classID);
  if (state!=-1) printf("  /\\ state_%d = %d ", classID,state);
  //-1 means no state
}

void start_ap_state_prime(int classID, int ap, int state, int j)
{
  if (ap==FULL)
    printf("  /\\ ap_%d_%d\' = %d /\\ tkwB_%d\' = 0 /\\ tkrB_%d\' = tkrB_%d - 1 /\\ tkr_%
         d_%d\' = 1 /\\ tkw_%d_%d\' = %d",classID,j,FULL,classID,classID,classID,classID,
         j,classID,j, K+1);
  else if (ap==PURE)
    printf("  /\\ ap_%d_%d\' = %d /\\ tkrB_%d\' = tkrB_%d - 1 /\\ tkr_%d_%d\' = 1",
         classID,j,PURE,classID,classID,classID,j);
}

void end_ap_state_prime(int classID, int ap, int state, int j)
{
  if (state!=-1)  printf("  /\\ state_%d\' = %d ",classID,state);
  if (ap==FULL)
      printf("  /\\ tkrB_%d' = tkrB_%d + 1 /\\ tkwB_%d\' = %d /\\ tkr_%d_%d\' = 0 /\\
           tkw_%d_%d\' = 0",classID,classID,classID, K+1,classID, j,classID, j);
  else if (ap==PURE)
      printf("  /\\ tkrB_%d' = tkrB_%d + 1  /\\ tkr_%d_%d\' = 0",classID,classID,classID,
           j);
}
```

```
void start_method(int classID,const char* name, int id, int ap, int state, int*
    ap_parameteres, int* st_parametres, int size, int j)
{
  printf(" start_%s_%d:\t", name,j);
  printf("  pc_%d_%d = %d ",classID, j, POST);
  start_ap_state(classID,ap,state,j);
  // if method does not have parameter, this code does not have any effect
 if (j==0) {
    int subscript=1;
    for (int i=0; i<size;i++) {
      subscript++;
      start_ap_state(subscript,ap_parameteres[i],st_parametres[i],j);
    }
  }
  printf("  ->");
  printf("  pc_%d_%d\' = %d /\\ method_%d_%d\' = %d",classID, j, PRE,classID, j, id);
  start_ap_state_prime(classID,ap,state,j);
  if (j==0) {
    int subscript=1;
    for (int i=0; i<size;i++) {
      subscript++;
      start_ap_state_prime(subscript,ap_parameteres[i],st_parametres[i],j);
    }
  }
  printf("\n");
}

void end_method(int classID,const char* name, int id, int ap, int state, int*
    ap_parameteres, int* st_parametres, int size, int j)
{
  printf(" end_%s_%d:\t", name,j);
  printf("  pc_%d_%d = %d /\\ method_%d_%d = %d",classID, j, PRE,classID, j, id);
  printf("   ->  ");
  printf("  pc_%d_%d\' = %d",classID,j, POST);
  end_ap_state_prime(classID,ap,state,j);
  if (j==0) {
    int subscript=1;
    for (int i=0; i<size;i++) {
      subscript++;
      end_ap_state_prime(subscript,ap_parameteres[i],st_parametres[i],j);
    }
  }
  printf("\n");
}

void create_alias(int classID,const char name[],  int j)
{
  printf(" create_alias_%s_%d:\t", name,j);
  printf("  ap_%d_%d = %d /\\ tkrB_%d > 0 /\\ state_%d != %d",classID,j, UNDEF,classID,
      classID,UNDEF);
  printf("  ->");
  printf("  pc_%d_%d\' = %d /\\ method_%d_%d\' = %d ", classID,j, POST,classID, j, UNDEF)
      ;
  printf("  /\\ ap_%d_%d\' = %d\n",classID, j, PURE);
}

void transitions_for_class_MttsTask(int j)
{
  start_constructor(MttsTask_CLASS,"MttsTask",MttsTask,-1,j);
  // -1 means, no required state
  start_method(MttsTask_CLASS,"setData",setData,FULL,CREATED,NULL,NULL,0,j);
  start_method(MttsTask_CLASS,"getData",getData,FULL,READY,NULL,NULL,0,j);
  start_method(MttsTask_CLASS,"execute",execute,FULL,READY,NULL,NULL,0,j);
  start_method(MttsTask_CLASS,"delete",DELETE,FULL,COMPLETE,NULL,NULL,0,j);

  end_constructor(MttsTask_CLASS,"MttsTask",MttsTask,CREATED,j);
  end_method(MttsTask_CLASS,"setData",setData,FULL,READY,NULL,NULL,0,j);
  //To introduce syntax error, comment out the line above, uncomment the line below
  //and do the necessary changes in the MttsTask.java
  //end_method(MttsTask_CLASS,"setData",setData,FULL,CREATED,NULL,NULL,0,j);

  end_method(MttsTask_CLASS,"getData",getData,FULL,READY,NULL,NULL,0,j);
  end_method(MttsTask_CLASS,"execute",execute,FULL,COMPLETE,NULL,NULL,0,j);
  end_method(MttsTask_CLASS,"delete",DELETE,FULL,DESTROYED,NULL,NULL,0,j);
  create_alias(MttsTask_CLASS,"MttsTask",j);
```

```
}

void transitions_for_class_MttsTaskDataX(int j)
{
  start_constructor( MttsTaskDataX_CLASS,"MttsTaskDataX",MttsTaskDataX,-1,j);
  // -1 means, no required state
  start_method(MttsTaskDataX_CLASS,"setID",setID,FULL,-1,NULL,NULL,0,j);
  start_method(MttsTaskDataX_CLASS,"setPriority",setPriority,FULL,-1,NULL,NULL,0,j);
  start_method(MttsTaskDataX_CLASS,"setDependencies",setDependencies,FULL,-1,NULL,NULL,0,
      j);
  start_method(MttsTaskDataX_CLASS,"getID",getID,PURE,-1,NULL,NULL,0,j);
  start_method(MttsTaskDataX_CLASS,"getPriority",getPriority,PURE,-1,NULL,NULL,0,j);
  start_method(MttsTaskDataX_CLASS,"getDependencies",getDependencies,PURE,-1,NULL,NULL,0,
      j);

  end_constructor(MttsTaskDataX_CLASS,"MttsTaskDataX",MttsTaskDataX,ALIVE,j);
  end_method(MttsTaskDataX_CLASS,"setID",setID,FULL,-1,NULL,NULL,0,j);
  end_method(MttsTaskDataX_CLASS,"setPriority",setPriority,FULL,-1,NULL,NULL,0,j);
  end_method(MttsTaskDataX_CLASS,"setDependencies",setDependencies,FULL,-1,NULL,NULL,0,j)
      ;
  end_method(MttsTaskDataX_CLASS,"getID",getID,PURE,-1,NULL,NULL,0,j);
  end_method(MttsTaskDataX_CLASS,"getPriority",getPriority,PURE,-1,NULL,NULL,0,j);
  end_method(MttsTaskDataX_CLASS,"getDependencies",getDependencies,PURE,-1,NULL,NULL,0,j)
      ;
  create_alias(MttsTaskDataX_CLASS,"MttsTaskDataX_CLASS",j);

}
void Trans(){
  // This array is used to store the access permissions for parameters
  int ap_parametres[5];
  // This array is used to store the states of parameters
  int st_parametres[5];

  printf("Transitions\n");
  for ( int j=0;j<=K;j++)
  {
    transitions_for_class_MttsTask(j);
    transitions_for_class_MttsTaskDataX(j);
  }
}


void Spec() {
  printf("Properties\n");

  //This checks the presence of deadlock.
  printf("  deadlock : !EX(true)\n\n");

  //The next batch checks for every method being able to execute.
  for (int i=0; i < NUM_CLASSES; i++) {
    for (int k=1; k<= NUM_METHODS[i]; k++){
      printf("  reach_method_%d_%d : EF(method_%d_0 = %d /\\ pc_%d_0 = %d)\n", i, k,  i,
          k, i, PRE);
    }

    //The adjacency matrix
    printf("\n\n");
    for (int k=0; k<=NUM_STATES[i]; k++) {
      for (int l=1; l<=NUM_STATES[i]; l++) {
        if (k!=l) {
        printf("  adjacent_%d_%d_%d : state_%d = %d /\\ EX(state_%d = %d)\n", i, k, l, i,
             k, i, l);
        }
      }
      printf("\n");
    }
  }
}

int main(int argc, char *argv[])
{
  K = 5;
  for (int i = 1; i < argc; ++i) {
    if (strcmp(argv[i], "-k")==0 && i<argc-1) {
      K = atoi(argv[i+1]);
      if (K<0) {
        fprintf(stderr, " ERROR: invalid number of references, %d.\n", K);
        fprintf(stderr, "        need at least 1 reference.\n");
        return 2;
      }
    }
  }

  Init();
  Trans();
  Spec();
  return 0;
}
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*
01- 08 - 2011

2. REPORT TYPE
Contractor Report

3. DATES COVERED *(From - To)*

4. TITLE AND SUBTITLE
Automated Verification of Specifications With Typestates and Access Permissions

5a. CONTRACT NUMBER
NNX08AC59A

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Siminiceanu, Radu I.; Catano, Nestor

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
534723.02.02.07.40

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)
NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
NASA/CR-2011-217170

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 64
Availability: NASA CASI (443) 757-5802

13. SUPPLEMENTARY NOTES
This report was prepared by the National Institute of Aerospace under subagreement number 27-001310 to SRI International, Menlo Park, CA, under NASA Cooperative Agreement NNX08AC59A.
Langley Technical Monitor: Benedetto L. Di Vito

14. ABSTRACT

We propose an approach to formally verify Plural specifications based on access permissions and typestates, by model-checking automatically generated abstract state-machines. Our exhaustive approach captures all the possible behaviors of abstract concurrent programs implementing the specification. We describe the formal methodology employed by our technique and provide an example as proof of concept for the state-machine construction rules. The implementation of a fully automated algorithm to generate and verify models, currently underway, provides model checking support for the Plural tool, which currently supports only program verification via data flow analysis (DFA).

15. SUBJECT TERMS
Access permissions; Formal methods; Model checking; Parallel programming languages

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Help Desk (email: help@sti.nasa.gov)
					19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	UU	36	(443) 757-5802