ATS Knowledge Documentation

Release 0.1

Zhiqiang Ren

June 01, 2015

Contents:

CHAPTER 1

Annotated ATS Programs

1.1 Goal

The normal way to start up a programming project using ATS is via imitating an existing example. The website serves as a collection of such examples along with intensive comments about the usage of ATS in these examples.

These examples come primarily from the Google group for ATS.

xxx

1.2 Examples

WireShark has been installed on all the Windows machines in the Instructional Lab (EMA 302). It is only usable during the period of the lab session.

1.2.1 Dining Philosopher with Animation (by Cairo)

Source File Download

Code

- dp_gui.dats

```
Copy from
Assignment 3:
Class: BU CAS CS520, Fall, 2013
Due: Thursday, the 26th of September, 2013
*)

(* ****** ****** *)
//
#include
"share/atspre_define.hats"
#include
```

- dp_observer.dats

```
(*
**
**
**
*)

(* ****** ****** *)

(*
Copy from
Assignment 3:
Class: BU CAS CS520, Fall, 2013
Due: Thursday, the 26th of September, 2013
*)

(* ****** ****** *)
//
#include
"share/atspre_define.hats"
#include
"share/atspre_staload.hats"
//
(* ****** ****** *)

staload
UN = "prelude/SATS/unsafe.sats"

(* ****** ****** *)

staload "libc/SATS/time.sats"

(* ****** ****** *)

staload "{$CAIRO}/SATS/cairo.sats"

(* ****** ****** *)


staload "{$GTK}/SATS/gdk.sats"
staload "{$GTK}/SATS/gtk.sats"
staload "{$GLIB}/SATS/glib.sats"
staload "{$GLIB}/SATS/glib-object.sats"

(* ****** ****** *)

staload "mythread.sats"

// staload "libc/SATS/stdlib.sats"
staload "libc/SATS/unistd.sats"


staload "dp_observer.sats"
dynload "dp_observer.dats"

staload "DiningPhil.sats"
dynload "DiningPhil.dats"
(* ****** ****** *)

```

```
%{^
typedef
struct { char buf[32] ; } bytes32 ;
%} // end of [%{^]
abst@ype bytes32 = $extype"bytes32"

(* ****** ****** *)

%{^
#define mystrftime(bufp, m, fmt, ptm) strftime((char*)bufp, m, fmt, ptm)
%} // end of [%{^]

(* ****** ****** *)

extern fun worker (darea: !GtkDrawingArea1): void

(* ****** ****** *)
val x: ref int = ref<int>(0)
val cg: ref double = ref<double>(0.0)
val cb: ref double = ref<double>(1.0)


(* ****** ****** *)

%{^
typedef char **charptrptr ;
%} ;
abstype charptrptr = $extype"charptrptr"


(* ****** ****** *)
(* ****** ****** *)
extern
fun{} fexpose (!GtkDrawingArea1): gboolean

implement{
} fexpose (darea) = let
  val () = draw_drawingarea (darea) in GFALSE
end // end of [fexpose]

extern fun{
} dp_anime_main (): void

(* ****** ****** *)

#define nullp the_null_ptr

#define W 400
#define H 400

implement{}
dp_anime_main
  ((*void*)) = () where
{
//
val win0 =
  gtk_window_new (GTK_WINDOW_TOPLEVEL)
val win0 = win0
val () = assertloc (ptrcast(win0) > 0)
val () = gtk_window_set_default_size (win0, (gint)W, (gint)H)
//
val opt = stropt_some"dining philosopher"
val issome = stropt_is_some(opt)
//
val () =
if issome then let
  val title = stropt_unsome (opt)
in
  gtk_window_set_title (win0, gstring(title))
end // end of [if] // end of [val]
//
val darea =
  gtk_drawing_area_new ()
val p_darea = gobjref2ptr (darea)
val () = assertloc (p_darea > 0)
val () = gtk_container_add (win0, darea)
//
val _sid = g_signal_connect
(
  darea, (gsignal)"expose-event", G_CALLBACK(fexpose), (gpointer)nullp
)
//
val _sid = g_signal_connect
(
  win0, (gsignal)"delete-event", G_CALLBACK(gtk_main_quit), (gpointer)nullp
)
val _sid = g_signal_connect
(
  win0, (gsignal)"destroy-event", G_CALLBACK(gtk_widget_destroy), (gpointer)nullp
)
//
val () = gtk_widget_show_all (win0)
//
val () = g_object_unref (win0) // HX: refcount of [win0] decreases from 2 to 1
//
// todo: Why does this pass the typechecking?
val () = mythread_create_cloptr (llam () => worker (darea))
val () = dp_init ()

//
val ((*void*)) = gtk_main ((*void*))

val () = g_object_unref (darea)
//
} // end of [dp_anime_main]

(* ****** ****** *)

implement
main0 (argc, argv) =
{
//
var argc: int = argc
var argv: charptrptr = $UN.castvwtp1{charptrptr}(argv)
//
val () = $extfcall (void, "gtk_init", addr@(argc), addr@(argv))

val ((*void*)) = dp_anime_main ((*void*))
//
} (* end of [main0] *)

implement worker (darea) = let
  val () = ignoret (sleep(1))
  //
  val (fpf_win | win) = gtk_widget_get_window (darea)
  //
  val isnot = g_object_isnot_null (win)
  //
  prval () = minus_addback (fpf_win, win | darea)
  //
  in
  //
  if isnot then let
    var alloc: GtkAllocation
    val () = gtk_widget_get_allocation (darea, alloc)
    val () = gtk_widget_queue_draw_area (darea, (gint)0, (gint)0, alloc.width, alloc.height)
  in
    worker(darea)
  end else
    ()
  // end of [if]
  //
end // end of [worker]

(* ****** ****** *)

(* end of [dp_gui.dats] *)
```

1.2.2 Operations pertaining to array0

Source File Download

Code

CHAPTER 2

ATS Tools' documentation

Contents:

2.1 Tag Generator For ATS-Anairiats

2.1.1 Description

For users of Emacs and Vim, it is common to browse source code spanning over multiple files with the support of *tag*. Here, we proivde a tool for generating *tag* files accepted by these two editors.

2.1.2 Download

Please download the file `ats-lang-tools.jar`.

2.1.3 Usage

The generation of *tag* file includes two steps as follows.

1. Use *atsopt* to collect information from the source files you have.

```
atsopt -o MYTAGS --taggen -s fact.sats -d fact.dats
```

2. Use *ats-lang-tools.jar* to generate the tag file from *MYTAGS*.

```
java -jar ats-lang-tools.jar --input MYTAGS -c --output tags # "c" for ctags with vim
java -jar ats-lang-tools.jar --input MYTAGS -e --output TAGS # "e" for etags with emacs
```

The next example shows how to generate the tag file for the source files of ATS-Postiats. Using option *–output-a*, *atsopt* would output to file *MYTAGS* acculaulatively, so we can combine *find* and *atsopt* to generate a large *MYTAGS*.

```
# Make sure we start from a clean slate.
rm -f MYTAGS

find ${PATSHOME}/src -name "*.sats" -exec atsopt --output-a MYTAGS --taggen -s {} \;
find ${PATSHOME}/src -name "*.dats" -exec atsopt --output-a MYTAGS --taggen -d {} \;
java -jar ats-lang-tools.jar --input MYTAGS -c --output tags
```

I use *${PATSHOME}/src* in the *find* command so that the generated *tags* would use absolute path for each file. In this way, we can open vim at any location.

The aforementioned method can be applied to ATS-Postiats as well. The following example shows how to generate the tag file for source files in the *prelude* of ATS-Postiats, which are written in ATS-Postiats.

```
PATH_PRELUDE=${PATSHOME}/prelude
MYTAGS_PATS_PRELUDE_PATS=${PATH_PRELUDE}/MYTAGS_PATS_PRELUDE_PATS

rm -rf ${MYTAGS_PATS_PRELUDE_PATS}
# Exclude two subdirectories "CODEGEN" and "DOCUGEN"
find ${PATH_PRELUDE}/SATS \( -name "CODEGEN" -o -name "DOCUGEN" \) -prune -o -name "*.sats" \
  -exec patsopt --output-a ${MYTAGS_PATS_PRELUDE_PATS} --taggen -s {} \;
find ${PATH_PRELUDE}/DATS \( -name "CODEGEN" -o -name "DOCUGEN" \) -prune -o -name "*.dats" \
  -exec patsopt --output-a ${MYTAGS_PATS_PRELUDE_PATS} --taggen -d {} \;

java -jar ${PATSHOME}/ats-lang-tools.jar -c --input ${MYTAGS_PATS_PRELUDE_PATS} --output ${PATH_
```

2.1.4 Development

The project is held on Github with the follwing address https://github.com/alex-ren/org.ats-lang.toolats.

2.2 Vim plug-in for ATS-Postiats

Please refer to this post for detailed information.

CHAPTER 3

ATS-Postiats' Blog

Some other sources of ATS: ATS Wiki.

Contents:

3.1 Syntax for *staload*

3.1.1 Macros in the path

In **ATS2**, keyword *staload* is followed by a literal string, which specifies the location of the file to be loaded. Besides those relative path and absolute path commonly seen, we can use *macro* inside the literal string. One example goes as follows:

```
staload "{$GTK}/SATS/gdk.sats"
```

The *GTK* is not a system environment variable. It is actually a macro defined in the ATS code. For example, we can use the following code to set the *GTK* to the directory containing **ATS2** code for gdk library.

```
#define GTK_targetloc "~/codebase/contrib/GTK"
staload "{$GTK}/SATS/gdk.sats"
```

Note: When we use *define* to define the macro, we are actually using the name *GTK_targetloc*, not *GTK*.

There are some special macros we can use, which are provided by the **ATS2** compiler, e.g. *PATSHOME* and *PATSHOMERELOC*. The following example shows their usage:

```
#define GTK_targetloc "$PATSHOMERELOC/contrib/GTK"
```

The macro *PATSHOMERELOC* in **ATS2** compiler is from the environment variable *PATSHOMERELOC*, which is set in the environment before executing *patscc* or *patsopt*.

Note: The macro *PATSHOMERELOC* is normally set to the *contrib* package released along with ATS.

Going further, let's have a look of the file **$PATSHOME/share/HATS/atspre_define_pdgreloc.hats**, which is included by the file **$PATSHOME/share/atspre_define.hats**, which is commonly included in almost all *dats* files.

```
//
#define
ZLOG_targetloc "$PATSHOMERELOC/contrib/zlog"
//
```

```
#define
JSONC_sourceloc "$PATSLIB_URL/contrib/json-c"
#define
JSONC_targetloc "$PATSHOMERELOC/contrib/json-c"
```

An ATS program using libraries of zlog and json-c would need to include the following snippet of code:

```
#include "share/atspre_define.hats"

staload "{$JSONC}/SATS/json_ML.sats"
```

3.2 Various *print* functions in ATS-Postiats

3.2.1 dddd

Todo

- `fprint_val<a>(stdout_ref, x)`
- `fprint_newline`

3.3 Miscellaneous topics related to ATS-Postiats

3.3.1 Setting of Environment

Please follow the instructions on ATS' website to install *ATS-Postiats* as well as *ATS2-contrib*. Here I just want to emphasize the setting of environment variables *PATSHOME* as well as *PATSHOMERELOC*. These two variables should be set in the environment before invoking the ATS compiler (*patscc* or *patsopt*). Also they are commonly referred in *Makefile* used in ATS projects. The first one (*PATSHOME*) should be set to the directory where ATS is installed. Normally I just download tarball for the source of ATS, decompress it, build ATS, and use built executables. (Simply put, I don't do `make install`.) Therefore I just set *PATSHOME* to the folder resulting from decompressing the tarball. *PATSHOMERELOC* should be set to the folder resulting for decompress the tarball for *ATS2-contrib*. Also the *PATH* variable should be set accordingly so that system can locate the compiler of ATS. Normally, I put these settings into one script file, say *pats.xxx.sh*. Then I do `source pats.xxx.sh` when I open a terminal for the first time. My *pats.xxx.sh* looks like the following:

```
# Script for setting environment for ATS-Postiats

PATSHOME=${HOME}/programs/ats2/ATS2-Postiats-0.0.8; export PATSHOME
PATSHOMERELOC=${HOME}/programs/ats2/ATS2-Postiats-contrib-0.0.7; export PATSHOMERELOC

PATH=${PATSHOME}/bin:${PATH}; export PATH
```

If you use `make install` to install ATS at the system level, your script would probably be like the following:

```
# Script for setting environment for ATS-Postiats

PATSHOME=/usr/local/lib/ATS2-Postiats-0.0.8; export PATSHOME
PATSHOMERELOC=${HOME}/programs/ats2/ATS2-Postiats-contrib-0.0.7; export PATSHOMERELOC
```

Note: Since we choose to install ATS at the system level, there's no need to set *PATH*.

3.3.2 Usage of *staload*

Some useful information about *staload* can be found in Wiki and my previous blog.

3.3.3 Standard *header* files

To use the *prelude* library of ATS, we would include the following code:

```
#include "share/atspre_define.hats"
#include "share/atspre_staload.hats"
```

To use the *ML* library of ATS, we would include the following code:

```
#include "share/HATS/atspre_staload_libats_ML.hats"
```

If we want to generate C code used on lower level systems, such as embedded system, we can replace these *header* files with appropriate ones to fit the targeting platform.

3.3.4 Type conversion

Todo

- `$UNSAFE.cast{natLt(n)}`

3.4 Targeting C# from program of ATS-Postiats

I am working on generating C# code from the second layer of the syntax tree of ATS-Postiats. The following includes some lessons I've learned and corresponding design decision I've made.

3.4.1 Convertion of types

Since C# supports powerful generic types, it feels so natural to try to translate the polymorphic types in ATS into generic types in C#. The following is the ATS code from which I want to generate C# code.

```
extern val p: ptr

fun foo1 {a:type} (x: (int, a)): int = x.0

fun test_foo1 (): void = let
  val x = foo1 ((0, p))
in
end

fun foo2 {a,b:type} (f: a -> b): int = 42

fun foo3 (): ptr -> ptr = let
  fun foo4 (x: ptr): ptr = x
in
  foo4
end

fun test_foo2 (): void = let
```

```
  val f = foo3 ()
  val x = foo2 (f)
in
end

/* ************ ************ */

fun foo5 (f: {a:type} a -> a): ptr = let
  val r = f (p)
in
  r
end
```

In ATS, tuple, record and function types have no names, and their equality is based on their content. In C#, all types must have names including structure, class, and delegate. Therefore I have to translate ATS code into C# code in the way shown in the sample code below. That is to create very general generic types in C# for tuple and functions types in ATS. It seems that there's no better way around this except the usage of *Object* type every where. Since I want to avoid *ugly* type casting, I plot to generate the code as follows

```
public class Ptr {};

/*
 * Tuple type must have name.
 */
public class Tuple2<T1, T2> {
    public T1 m1;
    public T2 m2;
    public Tuple2(T1 t1, T2 t2) {
        m1 = t1;
        m2 = t2;
    }
}

/*
 * Function type must have name.
 */
public delegate T2 Foo1<T1, T2>(T1 x);

public class Code {

    static public Ptr p = new Ptr();

    static public int foo1<T1>(Tuple2<int, T1> x) {
        return x.m1;
    }

    static public void test_foo1() {
        var x = foo1(new Tuple2<int, Ptr>(0, p));
    }

    static public int foo2<T1, T2>(Foo1<T1, T2> f) {
        return 42;
    }

    static public Foo1<Ptr, Ptr> foo3() {
        return foo4;
    }
    static public Ptr foo4(Ptr x) {
        return x;
    }

    static public void test_foo2() {
        var f = foo3();
        int x = foo2(f);
    }

    /* ******** ********* */

    // C# doesn't allow this.
    // static public Ptr foo5(Foo1 f) {
    //     var r = f(p);
    //     return r;
    // }

    static public void Main() {
        return;
    }
}
```

The code above compiles when function *foo5* commented out. The reason I have to comment out *foo5* goes as follows. In ATS, polymorphic function is of first class. (Object of polymorphic function type can be passed around. E.g. *foo5* takes one as input argument.) In C#, no object can be of open generic type (including generic delegate). Therefore the function "foo5" in ATS cannot be simply translated into a generic delegate of C#.

Therefore I choose to use *Object* type in C# to represent type parameter of polymorphic type in ATS. But this is still not good. Such candidate in C# is shown below.

```
using System;

public class Ptr {};

/*
 * Tuple type must have name.
 */
public class Tuple2<T1, T2> {
    public T1 m1;
    public T2 m2;
    public Tuple2(T1 t1, T2 t2) {
        m1 = t1;
        m2 = t2;
    }
}

/*
 * Function type mush have name.
 */
public delegate T Foo1<T>(T x);

public class Code {

    static public Ptr p = new Ptr();

    static public int foo1(Tuple2<int, Object> x) {
        return x.m1;
    }
    static public void test_foo1() {
        var x = foo1(new Tuple2<int, Object>(0, p));
    }

    static public int foo2(Foo1<Object> f) {
        return 42;
    }

    static public Foo1<Ptr> foo3() {
        return foo4;
    }

    static public Ptr foo4(Ptr x) {
        return x;
    }

    static public void test_foo2() {
        var f = foo3();
        int x = foo2((Foo1<Object>)(Object)f);
    }

    static public Ptr foo5(Foo1<Object> f) {
        var r = f(p);
        return (Ptr)r;
    }

    static public void Main() {
        return;
    }
}
```

Due to aforementioned decision, I have to give *foo2* the type *Foo1<Object>* as shown above. Then to make *test_foo2* compilable, I have to cast *f* to *Object*, then to *FOO1<Object>*. Also I have to use cast again in *foo5*. Such heavy usage of casting contradicts my original idea of relying on the generic type system of C#.

Therefore I simply choose to turn all the boxed types into *Object* and add proper type conversion whenever deemed necessary. (E.g. getting member of a tuple, invoking via a function pointer.) This is also the convention when generating C code from ATS program. The difference is that in C we rely on *void ** instead of *Object*. A hand written candidate is shown below.

```
using System;

/*
 * Tuple type must have name.
 */
public class Tuple2<T1, T2> {
    public T1 m1;
    public T2 m2;
    private Tuple2(T1 t1, T2 t2) {
        m1 = t1;
        m2 = t2;
    }
    static public Object create(T1 t1, T2 t2) {
        return new Tuple2<T1, T2>(t1, t2);
    }
}

/*
```

```
 * Function type mush have name.
 */
public delegate Object Foo1(Object x);

public class Code {

    static public Object p = new Object();

    static public int foo1(Object x) {
        return ((Tuple2<int, Object>)x).m1;
    }

    static public void test_foo1() {
        var x = foo1(Tuple2<int, Object>.create(0, p));
    }

    static public int foo2(Foo1 f) {
        return 42;
    }

    static public Foo1 foo3() {
        return foo4;
    }

    static public Object foo4(Object x) {
        return x;
    }

    static public void test_foo2() {
        var f = foo3();
        int x = foo2(f);
    }

    static public void Main() {
        return;
    }
}
```

In my implementation of C# code generator, I track the usage of all the tuples and records, define corresponding generic types for them. And I track all the function definitions, define corresponding delegate types for them.

3.5 Get value in the statics of ATS

Todo

organize

ATS-Postiats/prelude/basics_dyn.sats

```
dataprop
EQINT (int, int) = {x: int} EQINT (x, x)
//
extern prfun eqint_make {x,y:int | x == y} (): EQINT (x, y)
//
extern prfun
eqint_make_gint
```

```
  {tk:tk}{x:int} (x: g1int (tk, x)): [y: int] EQINT (x, y)

fun goo {x:int | x == 1} (): void = ()

fun foo (): void = let
  val x = 1
  val [y:int] EQINT () = eqint_make_gint (x)
in
  goo {y} ()
end
```

3.6 Template for Makefile of simple ATS project

An old version of Makefile template I wrote for ATS can be found here (todo).

The newer version of Makefile, which relies on *Makefile* provided by ATS, is shown below.

Makefile

```
#Makefile for untyped lambda calculus
######

include $(PATSHOME)/share/atsmake-pre.mk


######
#
CFLAGS += -O2
#
######
#
# TODO
#LDFLAGS :=  # uncomment this for a clean start of LDFLAGS
# LDFLAGS += -lm  # uncomment this for math library
#
# TODO
# Uncomment the following if you want the Boehm GC.
# By default, MALLOCFLAG is -DATS_MEMALLOC_LIBC
# LDFLAGS += -lgc
# MALLOCFLAG := -DATS_MEMALLOC_GCBDW
#
######

# TODO
# Add source files here.
SOURCES_DATS += UTLC.dats
SOURCES_SATS +=
SOURCES_C    +=

# TODO
# Specify the name of the target.
TARGET=UTLC

MYTARGET=$(TARGET)


include $(PATSHOME)/share/atsmake-post.mk
```

```
######

######

cleanats:: ; $(RMF) *_?ats.c

cleanall:: ; $(RMF) $(TARGET)

######

######


# For the future when tag generator is provided.
# .PHONY: tags
# tags:
#         $(RMF) tags MYTAGS
#         find ./ -name "*.sats" -exec $(PATSOPT) --output-a MYTAGS --taggen -s {} \;
#         find ./ -name "*.dats" -exec $(PATSOPT) --output-a MYTAGS --taggen -d {} \;
#         java -jar ../../../tool/ats-lang-tools.jar --input MYTAGS -c --output tags

###### end of [Makefile] ######
```

3.7 Global value with linear type

Global values with linear type can only be used in the global scope. They cannot be used inside a function scope. For example, the following code cannot pass the type checking.

```
staload
UNSAFE = "prelude/SATS/unsafe.sats"

absvtype VT

extern fun create (): VT
extern fun use (v: !VT): void

val v = create ()

fun temp (): void = let
  val () = use (v)
in
end

implement main0 () = ()
```

ATS compiler conplains that regexp_main.dats: 177(line=15, offs=17) – 178(line=15, offs=18): error(3): the linear dynamic variable [v$64(-1)] is expected to be local but it is not. regexp_main.dats: 177(line=15, offs=17) – 178(line=15, offs=18): error(3): a linear component of the following type is abandoned: [S2Ecst(VT)]. patsopt(TRANS3): there are [2] errors in total. exit(ATS): uncaught exception: _2home_2alex_2programs_2ats2_github_2ATS_2dPostiats_2src_2pats_error_2esats__FatalErrorExn(1025)

Usually we would cast the global value of linear type into non-linear type, and cast it back inside a function scope, which is shown below:

```
staload
UNSAFE = "prelude/SATS/unsafe.sats"

absvtype VT

extern fun create (): VT
extern fun use (v: !VT): void

val v = create ()

val ele = $UNSAFE.castvwtp0{ptr}(v)

fun temp (): void = let
  val v1 = $UNSAFE.castvwtp0{VT}(ele)
  val () = use (v1)
  prval ((*void*)) = $UNSAFE.cast2void (v1)
in
end

implement main0 () = ()
```

CHAPTER 4

ATS-Postiats' syntax

Contents:

4.1 Mapping from source code to data structure

4.1.1 Function declaration and implementation

We use the following source code as an example.

```
abstype ty (int, int)
extern fun foo {x: int} {y: int} (x: ty (x, y), y: int y):
  [q: int] int q

implement foo {x}{y}(x, y) = 3
```

For the function declaration, we would have a *D2Cdcstdecs*, which contains a list of *d2cst*. Each *d2cst* has type information of the defined constant. In this example, the type of function *foo* is quite complicated, which involves universal type (*S2Euni*) as well as existential type (*S2Eexi*).

ss

CHAPTER 5

Model Checking ATS

In this project, we focus on integrating model checking techniques seamlessly into the development of ATS program and ultimately build a practical system for verifying concurrent ATS program.

5.1 Tutorial

Before going to the details, let's have a quick look of how the methodology looks like. The producer-consumer problem is a classic one in field of concurrent programming. A recommanded implementation is described in the documentation of ATS [1], which exploits the type system of ATS to better ensure the correctness of the program. Certain mistakes, as stated below, can be avoided, which is great. However, a well-typed implementation for producer-consumer problem in ATS may still cause deadlock. And it's very difficult to soly rely on type system to capture such errors. Therefore, we start seeking help from other techniques, among which model checking is our pick here. It can help detect bugs related to temporal properties in concurrent systems and provide corresponding counterexamples. To apply the model checking technique, we need to form up the precise semantics of ATS programs, which in turn requires the precise semantics of those concurrency primitives related to communication and synchronization. We form up a collection of such primitives, based on which programers can build concurrent program in ATS with semantics meanful to the model checking techniques we employ here. Such collection is given in the file `conats.sats`. It also contains some other primitives used for model checking, which we shall explain as we see more examples.

The complete implementation for producer-consumer problem can be found here `16_reader_writer.dats`. You can also read, modify, and verify the implementation via our website Model Checking ATS. We illustrate some of the code snippets below.

As indicated in [1], a shared object contains a linear object, which in this example is a linear buffer. The primitives provided in `conats.sats` do not support such type. Therefore we define a linear type *lin_buffer* as well as corresponding functions for manupulating objects of such type, which is given below.

```
// Define linear buffer to prevent resource leak.
absviewtype lin_buffer (a:t@ype)

local
  assume lin_buffer (a) = atomref (a)
in
  fun lin_buffer_create {a:t@ype} (
    data: a): lin_buffer a = let
    val ref = conats_atomref_create (data)
  in
    ref
  end
```

[1] http://ats-lang.sourceforge.net/EXAMPLE/EFFECTIVATS/Producer-Consumer/main.html

```

    fun lin_buffer_update {a:t@ype} (
      lref: lin_buffer a, data: a): lin_buffer a = let
      val () = conats_atomref_update (lref, data)
    in
      lref
    end

    fun lin_buffer_get {a:t@ype} (
      lref: lin_buffer a): (lin_buffer a, a) = let
      val v = conats_atomref_get lref
    in
      (lref, v)
    end
  end
```

Three functions *conats_atomref_create*, *conats_atomref_update*, and *conats_atomref_get* are declared in `conats.sats`. Intuitively, they are used for creating a mutable object whose content can be accessed in an atomic manner.

In our example, we only need a linear buffer whose content is an integer. The following code defines the type *demo_buffer* for such linear buffer and some auxiliary functions for accessing it.

```
// Define linear integer buffer for demonstration.
viewtypedef demo_buffer = lin_buffer int

fun demo_buffer_isful (buf: demo_buffer): (demo_buffer, bool) = let
  val (buf, len) = lin_buffer_get (buf)
in
  (buf, len > 0)  // Assume the buffer can only hold 1 elements.
end

fun demo_buffer_isnil (buf: demo_buffer): (demo_buffer, bool) = let
  val (buf, len) = lin_buffer_get (buf)
in
  (buf, len <= 0)
end

fun demo_buffer_insert (buf: demo_buffer): demo_buffer = let
  val (buf, len) = lin_buffer_get (buf)
  val buf = lin_buffer_update (buf, len + 1)
in
  buf
end

fun demo_buffer_takeout (buf: demo_buffer): demo_buffer = let
  val (buf, len) = lin_buffer_get (buf)
  val buf = lin_buffer_update (buf, len - 1)
in
  buf
end
```

One thing worth mentioning is the number 1 we choose as the capacity of the virtual buffer shared by producer and consumer. In reality, a shared buffer may have a large capacity. But a big number may cause model checking not to be able to detect the potential bugs. Arguably, if our implementation is correct for a small capacity of shared buffer, it has better chances to be correct as well for large capacity.

Now we can create the linear buffer holding integer and then put it into a shared object which can be accessed by multiple threads. The corresponding code is shown below.

```
// Create a buffer for model construction.
val db: demo_buffer = lin_buffer_create (0)

// Turn a linear buffer into a shared buffer.
val s = conats_shared_create {demo_buffer}(db)
```

conats_shared_create is a function declared in `conats.sats`, whose semantics is about creating an shared object protecting its content via mutex and condition variables.

We now give out the code for producer and consumer. For the purpose of model checking, *producer* is actually a function which keeps increasing the counter inside the linear buffer whenever possible. If the capacity is reached, the producer would wait until the consumer takes out (by decreasing the counter) something out of the buffer. The same idea applies to the *consumer* functions. Notably, both *producer* and *consumer* would wake up the potentially waiting counterpart by sending a signal.

```
// Keep adding elements into buffer.
fun producer (x: int):<fun1> void = let
  val db = conats_shared_acquire (s)

  fun insert (db: demo_buffer):<cloref1> demo_buffer = let
    val (db, isful) = demo_buffer_isful (db)
  in
    if isful then let
      val db = conats_shared_condwait (s, db)
    in
      insert (db)
    end else let
      val (db, isnil) = demo_buffer_isnil (db)
      val db = demo_buffer_insert (db)
    in
      if isnil then conats_shared_signal (s, db)
      else db
    end
  end

  val db = insert (db)
  val () = conats_shared_release (s, db);
in
  producer (x)
end

// Keep removing elements from buffer.
fun consumer (x: int):<fun1> void = let
  val db = conats_shared_acquire (s)

  fun takeout (db: demo_buffer):<cloref1> demo_buffer = let
    val (db, isnil) = demo_buffer_isnil (db)
  in
    if isnil then let
      val db = conats_shared_condwait (s, db)
    in
      takeout (db)
    end else let
      val (db, isful) = demo_buffer_isful (db)
      val db = demo_buffer_takeout (db)
    in
      if isful then let
        // Omitting the following would cause deadlock
        // val db = conats_shared_signal (s, db)
      in db end
      else db
    end
  end

  val db = takeout (db)
  val () = conats_shared_release (s, db);
in
  consumer (x)
end
```

Due to the usage of linear type of ATS, ATS compiler would complain if a programmer forgets to call *conats_shared_acquire* to acquire the mutex (which is inside the shared object) before updating the counter, or *conats_shared_release* to release the mutex. However, type checking won't be able to detect the potential deadlock if the producer or consumer doesn't call the *conats_shared_signal* function.

Model checking can help detect the aforementioned bug. However, unlike type checking, model checking can only be applied to a runable program instead of a collection of functions. Therefore we set up the environment as follows so that we have a complete model. The model consists of two threads, one for producer and one for consumer. The *conats_tid_allocate* and *conats_thread_create* functions are provided by `conats.sats`. Intuitively, they are used for allocating thread id and creating new thread with a given function.

```
val tid1 = conats_tid_allocate ()
val tid2 = conats_tid_allocate ()

val () = conats_thread_create(producer, 0, tid1)
val () = conats_thread_create(consumer, 0, tid2)
```

Since model checking allows us to verify various properties of a program, we specify as follows that we want to verify that our program does not have deadlock.

```
%{$
#assert main deadlockfree;
%}
```

So far we have implemented the producer-consumer problem. With the appropriate implementations of functions from `conats.sats`, we can compile and run the ATS program. Due to the nondeterminism caused by concurrency, the potential deadlock may not happen during several runnings. But with model checking, we are guaranteed that there is no deadlock if our implementation can pass the model checking.

The model checking process goes as follows. We build a tool, which is able to extract a model from the ATS program given above. Currently, the extracted model is in the modeling langauge CSP#. We then use the state-of-art model checker PAT to check the generated model. To ease the whole process, we set up a website for readers to try this methodology on-line: Model Checking ATS. The aforementioned example can be found under the name "16_reader_writer.dats" in the dropdown list "Select ATS Example". We are working on building tools to better relate the model checking result (counterexample) to the original ATS program. However, it's still quite informative just by inspecting the current result of the model checker since the extracted model in CSP# is quite readable. As for the example, if we omit *conats_shared_signal* in *consumer*, model checking would give out the following result including the trace leading to the deadlock. (We omit the detail of the trace here for clarity purpose.)

```
=======================================================
Assertion: main() deadlockfree
********Verification Result********
The Assertion (main() deadlockfree) is NOT valid.
The following trace leads to a deadlock situation.
<init -> main_init -> main61_id_s1.0 -> lin_buffer_create_63_s1.0 -> main61_id_s2.0 -> ......
```

```
********Verification Setting********
Admissible Behavior: All
Search Engine: Shortest Witness Trace using Breadth First Search
System Abstraction: False


********Verification Statistics********
Visited States:2392
Total Transitions:4588
Time Used:0.3925891s
Estimated Memory Used:24059.904KB
```

Next we will illustrate more features of this methodology of combining type checking of ATS programming langauge with model checking technique to verify properties of concurrent programs.

5.2 Table of Contents

5.2.1 Ghosts for Model Checking

ATS programming language allows programmers to write code constructing proofs along side with operational code which does the actual computation. The code for proof is seen only by the type checker of ATS and is erased completely before the compiler of ATS starts generating executable programs. We extends such concept by allowing programmers to write code related to model checking. Such code can be seen by the type checker of ATS, which facilitate the type checking process, and will be erased (just like proof code) before the compiler of ATS starts generating executable programs. We call such code ghost code. Ghost code comes in various forms: Ghost function, Ghost value, Ghost type, and etc. We will talk about them in details when meeting related exampels.

Both operational code and ghost code have semantics to model checker we build. On the other hand, proof code is neglected by the model checker. The simplest but maybe the most stimulating example of ghost code is *mc_assert* as shown below.

```
fun foo (): void = let
  prval () = mc_assert (false)
in
end

val () = foo ()
```

The executable generated by ATS compiler from the code above simply does nothing since the only content of the function *foo* is a ghost function *mc_assert*, which is only meaningful to our model checker. Intuitively, we can view *mc_assert* as assertions normally used by programmers for runtime checking, but this time it's used at the stage of model checking.

Feel free to try the example using our online model checker. To use the model checker, we need to appending the following code to indicate that we want to verify that *mc_assert* succeeds.

```
%{$
#assert main |= G sys_assertion;
%}
```

The complete code (with some routine code for header files) for the example can be found here `demo_01_mc_assert.dats`

With *mc_assert*, we can use model checking as a test facility as shown in the following example.

```
typedef Int = [x:int] int x

fun sum {x:nat} (x: int x): Int = x * (x + 1) / 2

fun sum2 {x:nat}.<x>.(x: int x):<fun> Int =
  if x > 0 then x + sum2 (x - 1)
  else 0

val n = 3
val ret = sum (n)
prval mc_ret = sum2 (n)
prval () = mc_assert (ret = mc_ret)
```

In the example above, we have two versions of the summation function. We use model checking to verify that they generate the same output. *mc_ret* is a ghost value (declared via the keyword *prval*), and accordingly the executable generated by ATS compiler doesn't contain the invocation of *sum2*. However *sum2* itself is not a ghost function, and it can be used at runtime if programmers choose to. (Just for technical detail, currently only pure functions in ATS can be used to calculate ghost values. That's why the type of sum2 is a little bit verbose, since we prove that *sum2* always terminates.)

One thing worth mentioning is the type of *mc_assert* as shown below:

```
prfun mc_assert {b: bool} (x: bool b):<fun> [b == true] void
```

The type of the argument of *mc_assert* depends on a boolean value *b* in the statics of ATS. *mc_assert* assures the type system that *b* is true and as such the type checking of the whole program may be established successfully. Also the validity of such assertion is checking during the process of model checking.

5.2.2 Towards Concurrent Program

The type system of ATS consists of both dependent types and linear types. Please refer to ATS' documentation for its application in constructing verifiably correct sequential progarm. Linear types are of help to certain extent in ensuring safety properties about mutual exclusion. However, in general, the type system of ATS has difficulty in specifying properties of concurrent programs, e.g. invariants of objects across threads, absence of deadlock, liveness properties, and etc. Such incapability triggers our research in corporating model checking techniques into the verification process of ATS program.

Primitives for Concurrent Programming

But first of all, programmers have to be able to write concurrent programs in ATS. Currently, we add a set of primitives into ATS programming language to support concurrent programming in a style similar to using pthead. In the near future, we will add more primitives (e.g. channel, future) supporting different styles of concurrent programming.

The declarations of these primitives can be found in `conats.sats`. In the *Tutorial*, we already use *conats_shared_create*, *conats_shared_acquire*, *conats_shared_condwait*, *conats_shared_signal*, and *conats_shared_release*, whose types are declared as follows:

```
abstype shared_t (viewt@ype, int)
typedef shared (a:viewt@ype) = shared_t (a, 1)

fun conats_shared_create {a: viewt@ype} (ele: a): shared (a)

fun conats_shared_acquire {a: viewt@ype} {n:pos} (s: shared_t (a, n)): a
fun conats_shared_release {a: viewt@ype} {n:pos} (s: shared_t (a, n), ele: a): void

fun conats_shared_signal {a: viewt@ype} (s: shared (a), ele: a): a
fun conats_shared_condwait {a: viewt@ype} (s: shared (a), ele: a): a
```

conats_shared_create creates a shared object (of type *shared a*) holding a linear buffer. The concept of shared object is similar to that of monitor [2]. The types of the aforementioned functions guarantee they are invoked appropriately in a non-object-oriented language like ATS. (E.g. we have to call *conats_shared_acquire* before invoking *conats_shared_condwait* and *conats_shared_signal*. From the perspective of pthread programming, a shared object consists of a mutex and a condition variable working together for synchronization purpose.

However, sometimes it's not enough for a shared object to contain just one condition varialbe. For example, in *Tutorial*, the shared object has only one condition varialbe, which is used to indicate two situations: buffer is full or empty. The example has no deadlock because it only has one producer and one consumer. Simply adding one more consumer to the example would lead to potential deadlock. The complete code can be downloaded here `16_1_producer_consumer_m_1.dats`. You can also find this example at our website Model Checking ATS. One example of the potential deadlock can be thought as follows: Initially, the shared buffer is empty. Two consumers come and wait on the shared object. The producer comes and puts one element into the buffer and then wake up one consumer. However, the newly active consumer doesn't execute instantly. Instead, the producer comes again, tries to put another element into the buffer, and has to wait on the shared object since the capacity of the buffer is 1. Then the newly active consumer gets one element out of the buffer and wakes up another consumer. At this moment, the buffer is empty, the producer is waiting, and two consumers won't signal the shared buffer any more. And this leads to a deadlock. The counterexample found by the model checker by Breadth First Search confirms with our speculation.

To solve this problem, we also provide another version of shared object which contains multiple condition variables. The types of related functions are shown below.

```
abstype shared_t (viewt@ype, int)

fun conats_sharedn_create {a: viewt@ype} {n:pos} (ele: a, n: int n): shared_t (a, n)

fun conats_sharedn_signal {a: viewt@ype} {i,n:nat | i < n} (s: shared_t (a, n), i: int i, ele: a):

fun conats_sharedn_condwait {a: viewt@ype} {i,n:nat | i < n} (s: shared_t (a, n), i: int i, ele: a)
```

With such shared object, we can now set up two condition variables handling both full and empty buffers separately. The complete code can be download here `16_2_producer_consumer_m_1_2cond.dats`. A snappet of code for the producer is shown below.

```
// Keep adding elements into buffer.
fun producer (x: int):<fun1> void = let
  val db = conats_shared_acquire (s)

  fun insert (db: demo_buffer):<cloref1> demo_buffer = let
    val (db, isful) = demo_buffer_isful (db)
  in
    if isful then let
      val db = conats_sharedn_condwait (s, NOTEMP, db)
    in
      insert (db)
    end else let
      val (db, isnil) = demo_buffer_isnil (db)
      val db = demo_buffer_insert (db)
    in
      if isnil then conats_sharedn_signal (s, NOTFUL, db)
      else db
    end
  end
```

[2] http://en.wikipedia.org/wiki/Monitor_%28synchronization%29

```

    val db = insert (db)
    val () = conats_shared_release (s, db);
  in
    producer (x)
  end
```

In this implemenation, producer only signals the condition variable when the buffer is actually empty at that moment. This would lead to the missing of signal if we have multiple producers and consumers. (Please refer to section 8.2.2 of [3].) In the example `16_3_producer_consumer_m_m_signal.dats`, there is two producers, each of which inserts only one element, and two consumers, each of which takes out one element. And the problem of miss signal would lead to deadlock. The model checker would demonstrate this by a counterexample. One remedy is to sigal the condition variable every time. The other is to use *conats_sharedn_wait* instead of *conats_sharedn_signal*.

Bibliography

5.2.3 Accessing Global Ghost Variables

Global ghost variables

To facilite programmers to state the properties across the boundry of threads, we provide the concept of global ghost variables. (As being ghost, any flow of data from such variables to the operational state of the program is forbidden.) Programmers are also required to give identities to ghost variables, which provides a way to bridge model checking and type checking for program development. The following example demonstrates this concept.

```
stacst sid_init: sid
extern val mc_init: mc_gv_t sid_init

fun exec (x: int): void = let

  fun foo {init: pos}(pf: int_value_of (sid_init, init) | x: int): int = x

  prval (pf | init) = mc_get_int (mc_init)

  // mc_assert cannot be omitted though it is ghost code.
  prval () = mc_assert (init > 0)

  val _ = foo (pf | x)
in
end

val tid1 = conats_tid_allocate ()

val () = conats_thread_create(exec, 0, tid1)

prval () = mc_set_int (mc_init, 1)


%{$
// #assert main deadlockfree;

#assert main |= G sys_assertion;

%}
```

[3] The Art of Multiprocessor Programming

We explain this example in details as follows:

```
stacst sid_init: sid
extern val mc_init: mc_gv_t sid_init
```

stacst declares a constant *sid_init* in the *statics* of ATS. This contant serves as the identifier of a ghost variable. *extern val* declares a value *mc_init*, which is the counterpart of *sid_init* in the dynamics of ATS. (In the future, only *stacst* is needed after we simplify the model generation process.)

```
fun foo {init: pos}(pf: int_value_of (sid_init, init) | x: int): int = x
```

The type of function *foo* states that a proof that ghost variable *sid_init* used to be positive is needed to invoke the function. Therefore, if we erase the ghost code *mc_assert* from the following code

```
fun exec (x: int): void = let

  fun foo {init: pos}(pf: int_value_of (sid_init, init) | x: int): int = x

  prval (pf | init) = mc_get_int (mc_init)

  // mc_assert cannot be omitted though it is ghost code.
  prval () = mc_assert (init > 0)

  val _ = foo (pf | x)
in
end
```

the type checker of ATS would complain that there exists unsolved constraint in the type checking process. From the example, we can see that a set of well designed interfaces for functions can force programmers to incorporate model checking method during the development process. The complete program can be downloaded here `24_global_ghost_variable.dats`. Because of the following code

```
val tid1 = conats_tid_allocate ()

val () = conats_thread_create(exec, 0, tid1)

prval () = mc_set_int (mc_init, 1)
```

The program stands a chance of making the *mc_assert* (in function *exec*) to fail since the ghost variable is set to 1 after creating a new thread to execute function *exec*. The model checking process can help us detect such problem.

Atomicity in ghost code

For specification purpose, sometimes it's necessary to group sereral ghost code into one atomic step. We provide two ghost primitives *mc_atomic_start* and *mc_atomic_end* to mark the scope for an atomic step, which we call an atomic scope, consisting of both ghost code and operational code. The following program shows the usage of the atomic step.

```
  stacst mid: sid

  extern val mc_m: mc_gv_t mid

  fun foo1 (): void = let
    prval () = mc_atomic_start()
    prval () = mc_set_int (mc_m, 3)
    prval () = mc_set_int (mc_m, 4)
    prval () = mc_atomic_end()
  in
  end
```

```

fun foo2 (x: int): void = let
  prval (pf | x) = mc_get_int (mc_m)
  prval () = mc_assert (x <> 3)
in
end

val tid1 = conats_tid_allocate ()

val () = conats_thread_create(foo2, 0, tid1)

val () = foo1 ()
```

The *mc_assert* in function foo2 succeeds because the state in which the ghost variable *mc_m* is set to 3 is not observable by other threads. The complete code can be downloaded here `18_atomic_opr.dats`.

Currently, it's programmers' responsibility to make sure that only primitives for accessing ghost variables (*mc_set_int*, *mc_get_int*) and primitives for accessing global references (*conats_atomref_update*, *conats_atomref_get*, *conats_atomarrayref_update*, *conats_atomarrayref_get*) can appear in an atomic scope. Also they have to make sure that an atomic scope can contain at most one operational primitive for accessing global reference while there's no limit for the number of primitives for accessing ghost variables.

5.2.4 Virtual Lock

Improper handling of resources (e.g. memory) in a program may lead to various bugs (e.g. memroy leak) in sequential programs. The problem gets even worse when entering the concurrent domain, in which simultaneous access to shared resource by multiple threads is feasible. One example is that we may lose the integrity of data if two threads are using a shared memory to transfer data. Techniques for solving this problem generally rely on mutual exclusion principles to control access to shared resources. Mutual exclusion introduces a measure of synchronization, but with the cost of losing efficiency. With a deliberate design, sometimes we can remove the need for synchronization while maintaining the desired property of mutual exclusion. Simpson's four-slot fully asynchronous communication mechanism [4] demonstrates such idea. However, it's very difficult to verify that the deemed mutual exclusion property actually holds in the design. To tackle this problem, we provide two primitives supporting the concept of "virtual lock" to allow programmers to specify assumptions of mutual exclusion to various granularities according to their design. And such assumption can then be verified by our model checker.

Let's illustrate the usage of "virtual lock" using the following example of two-slot mechanism. Consider the scenario in which one writer and one reader try to communicate via a shared resource consisting of multiple memory regions (two in this example). Due to hardware constraint, access to each memory region cannot be done atomically. Therefore, reader may get inconsistent data if writer is writing the same region at the same time. The following code shows the proposed types for the shared resource (*dataslots_t*) as well as the interfaces for accessing it.

```
abstype dataslots_t (t@ype, int)

absviewtype own_slot_vt (int)

fun dataslots_create {a:t@ype} {x:pos} (
  x: int x, v: a): dataslots_t (a, x)

fun dataslots_update {a:t@ype} {x,i:nat | i < x}
  ( vpf: own_slot_vt (i)
  | slots: dataslots_t (a, x), i: int i, v: a
  ): (own_slot_vt i | void)

fun dataslots_get {a:t@ype} {x,i:nat | i < x}
```

[4] H.R. Simpson, Four-slot fully asynchronous communication mechanism

```
    ( vpf: own_slot_vt (i)
    | slots: dataslots_t (a, x), i: int i
    ): (own_slot_vt i | a)
```

The usage of linear type *own_slot_vt* states clearly that *dataslots_update* and *dataslots_get* require mutual exclusion on the memory region to be accessed. Normally, programmers ensure such property by the usage of synchronization primitives (e.g. mutex). However, in the following code, we try to gain mutual exclusion by the usage of a few global variables the access for which is atomic. The code is shown below.

```
typedef data_t = dataslots_t (int, 2)
val data: data_t = dataslots_create (2, 0)

typedef int2 = [i: int | i >= 0 && i <= 1] int i

// control variables
val latest = conats_atomref_create {int2} (0)

fun write (item: int): void = let
  val index = 1 - conats_atomref_get (latest)

  prval vpf = mc_acquire_ownership (index)
  val (vpf | _) = dataslots_update (vpf | data, index, item)
  prval () = mc_release_ownership (vpf)

  val () = conats_atomref_update (latest, index)
in
end

fun read (): int = let
  val index = conats_atomref_get (latest)

  prval vpf = mc_acquire_ownership (index)
  val (vpf | item) = dataslots_get (vpf | data, index)
  prval () = mc_release_ownership (vpf)
in
  item
end
```

In the example, the shared resource (*data_t*) contains two regions (slots). *lastest* is a global reference for an integer, which is created by the primitive *conats_atomref_create*. (Primitives *conats_atomref_create*, *conats_atomref_get*, and *conats_atomref_update* are provided as an extension to ATS to support concurrent programming.) To pass the type checking of ATS, we use two functions *mc_acquire_ownership* and *mc_release_ownership* to generate and destroy the linear ghost value (*vpf*), which serves as the warranty for mutual exclusion. *mc_acquire_ownership* and *mc_release_ownership* are not primitives. Instead, they are user-defined ghost functions. Their implementation is shown below.

```
  prfun mc_acquire_ownership .<>. {i: nat}
    (i: int i): own_slot_vt (i) = mc_vlock_get (i, 0, 1, 1)

  prfun mc_release_ownership .<>. {i: nat}
    (vpf: own_slot_vt (i)): void = mc_vlock_put (vpf)
```

The two ghost functions are built upon two primitives *mc_vlock_put* and *mc_vlock_get*. Intuitively, *mc_vlock_get (x, y, a, b)* indicates the acquision of a *virtual lock* covering a rectangle with *(x, y)* as the upper left corner, *a* as the width (x-axis), and *b* as the height (y-axis), and *mc_vlock_put* indicates the release of the lock. And our model checker would check that under no circumstances would two threads try to acquire two virtual locks covering overlapping areas simutaneously. And this serves as the verification of mutual exclusion. To model checking the example, we would need to add the following code to implement those interfaces for accessing shared resource.

```
fun dataslots_create {a:t@ype} {x:pos} (
  x: int x, v: a): dataslots_t (a, x) =
   conats_atomarrayref_create {a} (x, v)

fun dataslots_update {a:t@ype} {x,i:nat | i < x}
  ( vpf: own_slot_vt (i)
  | slots: dataslots_t (a, x), i: int i, v: a
  ): (own_slot_vt i | void) = let
  val () = conats_atomarrayref_update (slots, i, v)
in
  (vpf | ())
end

fun dataslots_get {a:t@ype} {x,i:nat | i < x}
  ( vpf: own_slot_vt (i)
  | slots: dataslots_t (a, x), i: int i
  ): (own_slot_vt i | a) = let
  val v = conats_atomarrayref_get (slots, i)
in
  (vpf | v)
end
```

In the aforementioned code, this implementation is actually based on the primitives for creating and accessing array. This is not necessary if our focus is to verify the validity of mutual exclusion.

The complete code can be downloaded here `20_1_two_slot_acm.dats`. Without much thinking, we know that this implementation cannot pass model checking since writer just switches between two slots. Going further, the implementation (`20_2_three_slot_acm.dats`) using three slots doesn't work either. And based on the implementation using four slots (`20_3_four_slot_acm.dats`), we verify that Simpson's four-slot asynchronous mechanism possesses the acclaimed mutual exclusion property.

Bibliography

5.3 Bibliography

CHAPTER 6

Indices and tables

- genindex
- modindex
- search