

Bounds Tutorial

Antti Siirtola

April 30, 2013

1 Introduction

Software systems have natural parameters, like the number of replicated parts and the size of parametric data types, which can take infinitely many values. Consequently, all software systems can be considered as multiparametrised finite-state machines where the parameters determine the (maximum) number of replicated components (like objects), their relationships and the size of basic data types. That is why the question on the correctness of a software system is naturally expressed as the parametrised verification problem: given a parametrised system implementation and specification, determine whether the implementation is correct with respect to the specification for all parameter values.

Bounds2 is a tool that enables parametrised verification by establishing upper bounds, i.e., *cut-offs*, for the values of parameters such that if there is a bug in an implementation instance with a parameter value greater than the cut-off, then there is an analogous bug in an implementation instance where the values of the parameters are within the cut-offs. When using Bounds2, implementations and specifications are composed of labelled transitions systems (LTSs) with explicit input and output events by using parallel composition and hiding. One can also use several kinds of parameters: *process types* represent the sets of the identifiers of replicated components, *data types* denote the sets of data values, *typed variables* refer to the identities of individual components or data values and *relation symbols* represent binary relations over process types. Correctness is understood either as a refinement, which can be trace inclusion [1] or alternating simulation [2], or the compatibility of the components of the implementation [2].

The tool consists of two parts. The instance generator determines cut-offs for the types, computes the allowed parameter values up to the cut-offs and outputs the corresponding finite state verification tasks. After that, the outputted instances are verified by an instance checker specific to the notion of correctness. Trace refinement and compatibility checkers exploit the refinement checker FDR2 [1] to verify the instances, the alternating simulation checker makes use of MIO Workbench [3] refinement checker, too.

2 Installation

To install and use Bounds you need a number of other software packages.

1. Bounds uses nauty (No AUTomorphisms?, Yes!) to detect and remove isomorphic parameter values. Nauty package is downloaded automatically by the configuration script, so a working network connection is needed during the first time installation.
2. ANTLR3 (ANother Tool for Language Recognition) C v3.2 development files must be installed, v3.4 is buggy and hence not supported. A package containing the library and headers can be downloaded from <http://www.antlr.org/download/C>. Alternatively, one can install the debian package `libantlr3c-dev`.
3. Google's tcmalloc library is needed for more efficient memory management. A package containing the library can be downloaded from <http://code.google.com/p/gperftools>. Alternatively, one can install the debian package `libgoogle-perftools-dev`.
4. Bounds produces FDR2 (Failures-Divergences Refinement) compatible output. Hence, in order to complete the verification process, FDR2 must be properly installed as well. FDR2 is needed by `boundsTraceRefinement|Compatibility|AlternatingSimulationChecker` scripts. FDR2 can be downloaded from http://www.fsel.com/fdr2_download.html.
5. `boundsAlternatingSimulationChecker` exploits MIO Workbench, too, which is an Eclipse plugin. MIO Workbench is downloaded automatically by the configuration script, so a working network connection is needed during the first time installation.
6. Eclipse is not installed automatically. It is recommended to install Eclipse 3.7 (Indigo) with Modelling Tools manually from <http://www.eclipse.org/downloads/packages/release/indigo/sr2>, not automatically through a package manager, at least Eclipse installed from an Ubuntu package does not recognise external plugins properly. After installing Eclipse, you need to install XText modelling component via Eclipse's Help menu, too.

Having prepared all the auxiliary software as explained above, run

```
./configure  
make  
make install
```

to compile and install Bounds.

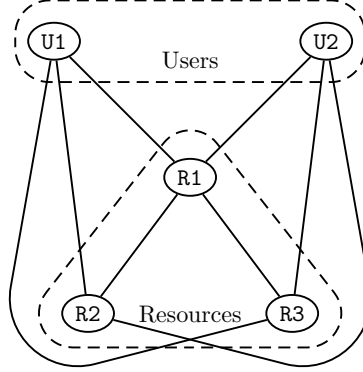


Figure 1: An instance of SRS with two users and three resources and connections between the components

3 Case: Shared Resource System (SRS)

The input language of Bounds is called Communicating Parameterised Systems (CPS) according to its role model CSP (Communicating Sequential Processes) [1]. CPS is basically the formalism presented in [4] with some syntactic sugar to enable the compact and comfortable representation of large processes.

To introduce the input language of Bounds, CPS, and the tool itself, you should consider a shared resource system (SRS) with an arbitrary number of users competing for an access to an arbitrary number of shared resources arranged in the form of a forest (Fig. 1). A user gets a read (write) access to a subtree of resources after successfully requesting a read lock `sr` (write lock `sw`) on the root of the subtree and a weaker read intention lock `ir` (write intention lock `iw`) on all its ancestors. The purpose of the read (write) intention lock is to indicate the existence of the read (write) lock deeper in the tree. Several users can hold a lock on a resource simultaneously only if all of them have either intention locks or read related locks (Fig. 2). Our goal is to formally model the system implementation and the specification and prove that in our construction it is not possible for a user to access a resource if somebody else is writing to it.

3.1 Introduce a Type for Each Class of Components

There are two kinds of components, resources and users, so we introduce types `R` and `U` to represent the sets of their identities, respectively.

```
type R
type U
```

The types `R` and `U` denote sets $\{R_1, R_2, \dots, R_k\}$, where $k \in \mathbb{Z}_+$, and $\{U_1, U_2, \dots, U_n\}$, where $n \in \mathbb{Z}_+$, respectively.

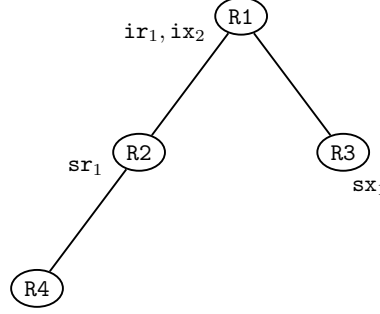


Figure 2: User U1 has a read access to Resources R2 and R4 (i.e. the read lock **sr** at the root of the subtree that contains R2 and R4, the subscript of the lock indicates its owner) and the read intention lock **ir** at the root, User U2 has a write access to (i.e. lock **sx** on) Resource R3 and the write intention lock **ix** at the root

3.2 Introduce Relation Symbols and Variables to Represent Non-Trivial Relationships Between the Components

The relationship between the resources is represented using a relation symbol **Anc**.

```
rels Anc: R,R
```

The relationship between the users (the empty relation) and the relationship between the users and the resources (the maximal bipartite topology) are trivial and can be handled without the use of relation symbols.

We also need some variables to refer to the identities of individual users and resources when describing system and specification behaviour, and the values of **Anc**.

```
var u : U
var u1 : U
var u2 : U
var r : R
var r1 : R
var r2 : R
var r3 : R
```

3.3 Introduce Communication Events

In CPS, communications are called events and they consists of a channel and a data part. For SRS, we define six channels for locking events and two channels for both read and write events.

```

chan srlock : U,R
chan sxlock : U,R
chan irlock : U,R
chan ixlock : U,R
chan unlock : U,R
chan nolock : U,R
chan rdbeg  : U,R,R
chan rdend  : U,R,R
chan wrbeg  : U,R,R
chan wrend  : U,R,R

```

The locking events consists of a user identifier and a resource identifier. For example, `srlock(u,r)` indicates that the user `u` puts a subtree read lock `sr` on the resource `r`.

The read and write events, in turn, consists of a user identifier and two resource identifiers. For example, `rdbeg(u,r2,r1)` indicates that the user `u` starts reading a resource `r2` based on the lock it has on the resource `r1`.

3.4 Specify the Allowed Values of Variables and Relation Symbols

The system implementation and specification will be first modelled from the viewpoint of two resources, where one resource is an ancestor of the other. That is why we define `Anc` as a proper ancestor relation, i.e. the transitive closure of a forest. (Here, an ancestor of a resource can also be the resource itself and a proper ancestor refers to an ancestor other than the resource itself.)

The values of variables and relation symbols are expressed in the universal fragment of first order logic (FOL), where terms are restricted to variables and predicated to relation symbols. (Also existential quantification can be used but then the tool is not able to compute the cut-offs automatically.) In CPS, formulae of this kind are called valuation constraints and specified using `vc` keyword.

```

// Anc is irreflexive
vc irrefl = \r : !r Anc r
// Anc is transitive
vc trans = \r1, r2, r3 : r1 Anc r2 & r2 Anc r3 -> r1 Anc r3
// the ancestors of a resource must be related by Anc
vc ancprop = \r1, r2, r3 :
    !r1 = r2 & r1 Anc r3 & r2 Anc r3 ->
        r1 Anc r2 | r2 Anc r1
// the transitive closure of a forest
vc ForestTopo = irrefl & trans & ancprop

```

Variables introduced earlier will not occur free in the final system and specification, i.e. they have no special system level meaning like the root of a tree,

but they are always bound by the universal quantifier (\forall), the replicated parallel composition or the generalised set union ($(_)$). That is why there is no need to restrict their values here.

3.5 Capture the Behaviour of the System Implementation and the Specification from the Viewpoint of Finitely Many Components in Finitely Many LTSs

To represent parameterised systems in our formalism, you need to use compositional modelling style [5]. That is, the behaviour of the system implementation is first captured in finitely LTSs, each of which represents the behaviour of the system from the viewpoint of finitely many components. After that, propositional conditions are used to restrict the values of variables and the parts are put together using standard and replicated parallel compositions. The specification is modelled similarly. Finally, the events irrelevant to the specification are hidden in the system.

We build the system model from three LTSs each of which represents the behaviour of the implementation from the viewpoint of two or three users and resources.

User1 captures the behaviour of a user u from the viewpoint of reading and writing to a resource $r2$ based on the lock on its ancestor $r1$. The LTS **User1** formally says that if u puts a read (write) lock on $r1$, then u can read (respectively write to) the resource $r2$. This view is obviously needed because we are interested in read and write operations a user makes.

User2 captures the behaviour of a user u from the viewpoint of locking a resource $r2$ and its proper ancestor $r1$. The LTS **User2** formally says that a read or read intention lock (respectively a write or write intention lock) can be requested to $r2$ only after the successful request of a read (respectively write) intention lock on $r1$. Moreover, **User2** ensures that $r1$ can be unlocked only if $r2$ is not locked or after $r2$ has been unlocked. This view to the system is needed because the lock requests a user makes on a resource and its ancestors are interdependent.

Lock stores the lock of a user $u1$ on a resource r and restricts the requests of another user $u2$ accordingly. If $u1$ has an intention lock or a read-related lock on r , then $u2$ can successfully request respectively intention or read-related locks, but if $u1$ has the write lock on r , then no request of $u2$ is allowed. This view is needed because also the lock requests of distinct users on the same resource are interdependent.

In CPS, parameterised LTSs are specified using the `plts` keyword.

```
plts User1 =
  lts
    I =    srlock(u,r1) -> R
```

```

        [] sxlock(u,r1) -> W
        [] tau() -> I
    R =    rdbeg(u,r2,r1) -> R2
        [] srlock(u,r1) -> R
        [] sxlock(u,r1) -> W
        [] unlock(u,r1) -> I
        [] tau() -> R
    R2 =    rdend(u,r2,r1) -> R
    W =    rdbeg(u,r2,r1) -> W2
        [] wrbeg(u,r2,r1) -> W3
        [] srlock(u,r1) -> W
        [] sxlock(u,r1) -> W
        [] unlock(u,r1) -> I
        [] tau() -> W
    W2 =    rdend(u,r2,r1) -> W
    W3 =    wrend(u,r2,r1) -> W
from I

plts User2 =
    lts
        I =    irlock(u,r1) -> IR
            [] srlock(u,r1) -> I
            [] ixlock(u,r1) -> IX
            [] sxlock(u,r1) -> I
            [] noload(u,r1)-> UN
            [] unlock(u,r2) -> UN
            [] noload(u,r2) -> UN
        IR =    irlock(u,r2) -> I
            [] srlock(u,r2) -> I
            [] tau() -> I
        IX =    ixlock(u,r2) -> I
            [] sxlock(u,r2) -> I
            [] tau() -> I
        UN =    unlock(u,r1) -> I
            [] tau() -> I
from I

plts Lock =
    lts
        NO =    irlock(u1,r) -> IR
            [] srlock(u1,r) -> SR
            [] ixlock(u1,r) -> IX
            [] sxlock(u1,r) -> SX
            [] unlock(u1,r) -> NO
            [] noload(u1,r) -> NO
            [] irlock(u2,r) -> NO

```

```

        [] srlock(u2,r) -> NO
        [] ixlock(u2,r) -> NO
        [] sxlock(u2,r) -> NO
IR =   irlock(u1,r) -> IR
        [] srlock(u1,r) -> SR
        [] ixlock(u1,r) -> IX
        [] sxlock(u1,r) -> SX
        [] unlock(u1,r) -> NO
        [] irlock(u2,r) -> IR
        [] srlock(u2,r) -> IR
        [] ixlock(u2,r) -> IR
SR =   irlock(u1,r) -> SR
        [] srlock(u1,r) -> SR
        [] ixlock(u1,r) -> SX
        [] sxlock(u1,r) -> SX
        [] unlock(u1,r) -> NO
        [] irlock(u2,r) -> SR
        [] srlock(u2,r) -> SR
IX =   irlock(u1,r) -> IX
        [] srlock(u1,r) -> SX
        [] ixlock(u1,r) -> IX
        [] sxlock(u1,r) -> SX
        [] unlock(u1,r) -> NO
        [] irlock(u2,r) -> IX
        [] ixlock(u2,r) -> IX
SX =   irlock(u1,r) -> SX
        [] srlock(u1,r) -> SX
        [] ixlock(u1,r) -> SX
        [] sxlock(u1,r) -> SX
        [] unlock(u1,r) -> NO
from NO

```

To formalise the specification, note that every illegal behaviour can be traced back to two different users $u1$, $u2$ that write to a resource $r3$ simultaneously based on the locks they have on respectively some ancestors $r1$, $r2$. Therefore, we first capture the specification in an LTS **Prop2** from the viewpoint of $u1$, $u2$ accessing $r3$ based on the locks on respectively $r1$, $r2$. However, to correctly present the specification in the presence of a single user only, we also introduce an LTS **Prop1**, which captures the specification from the viewpoint of $u1$ accessing $r2$ based on the lock on $r1$.

```

plts Prop2 =
  lts
    NO =   rdbeg(u1,r3,r1) -> R1
          [] rdbeg(u2,r3,r2) -> R2
          [] wrbeg(u1,r3,r1) -> N1
          [] wrbeg(u2,r3,r2) -> N2

```



```

      N1 =   wrend(u1,r3,r1) -> NO
      N2 =   wrend(u2,r3,r2) -> NO
      R1 =   rdend(u1,r3,r1) -> NO
            [] rdbeg(u2,r3,r2) -> RR
      R2 =   rdend(u2,r3,r2) -> NO
            [] rdbeg(u1,r3,r1) -> RR
      RR =   rdend(u1,r3,r1) -> R2
            [] rdend(u2,r3,r2) -> R1
from NO

plts Prop1 =
  lts
    NO =   rdbeg(u,r2,r1) -> N1
          [] wrbeg(u,r2,r1) -> N2
    N1 =   rdend(u,r2,r1) -> NO
    N2 =   wrend(u,r2,r1) -> NO
from NO

```

You can also use fixed sets and relations/functions over these sets to represent LTSs in a compact and convenient way. To see how to use these constructs, see taDOM2+ examples in `examples/tadom2+_repeatable.cps` [6, 7].

3.6 Introduce a Propositional Guard for Each LTS if Needed

- In `User1` and `Prop1`, `r1` denotes an ancestor `r2`, so the LTSs are enclosed within a propositional condition

$$\text{vc AncEq12} = (\text{r1} = \text{r2} \mid \text{r1 Anc r2})$$

- In `User2`, `r1` represents a proper ancestor of `r2`, so the LTS `User2` is enclosed within a propositional formula

$$\text{vc Anc12} = (\text{r1 Anc r2})$$

- In `Lock`, `u1` and `u2` denote different users, so `Lock` is put inside a propositional formula

$$\text{vc Neq12} = (!\text{u1} = \text{u2})$$

- In `Prop2`, `u1` and `u2` denote different users and `r1`, `r2` ancestors of `r3`, so we enclose `Prop2` within a propositional formula `AncEq13 & AncEq23 & Neq12` where

$$\begin{aligned} \text{vc AncEq13} &= (\text{r1} = \text{r3} \mid \text{r1 Anc r3}) \\ \text{vc AncEq23} &= (\text{r2} = \text{r3} \mid \text{r2 Anc r3}) \end{aligned}$$

3.7 Compose the LTSs in Parallel

To build the model of the system implementation and the formal specification, you should enclose the guarded LTSs within replicated parallel compositions such that every variable, which does not have a specific system level meaning, becomes bound. Finally, you compose the parts together using the standard parallel composition.

```
plts Sys = (|| u, r1, r2 : [PrecEq12] User1) ||
           (|| u, r1, r2 : [Prec12] User2) ||
           (|| r, u1, u2 : [Neq12] Lock)
plts Spec = (|| u, r1, r2 : [PrecEq12] Prop1) ||
           (|| u1, u2, r1, r2, r3 :
            [PrecEq13 & PrecEq23 & Neq12] Prop2)
```

3.8 Hide the Events Irrelevant to the Specification

In CPS, the sets of parameterised events are specified using `pset` keyword. In our case, we hide the events related to locking, denoted by `LockAct`. Hence, the system implementation gets the form `Sys \ LockAct`.

3.9 Specify the Parameterised Verification Task

The correctness of the SRS implementation can be now stated as the question whether `Sys \ LockAct` is a traces refinement of `Spec` whenever `ForestTopo` is satisfied. In CPS, this is specified in the following way.

```
trace refinement: verify Sys \ LockAct against Spec when ForestTopo
```

3.10 Run the Instance Generator

To reduce a parameterised verification task into (finitely many) finite-state verification tasks, just give the CPS file as a parameter to the `Bounds` instance generator.

```
boundsInstanceGenerator examples/SRS.cps
```

After reading the file, the instance generator reports the cut-off sizes for the types. As types are used to specify the domains of variables and relation symbols, these cut-offs implicitly limit the values of other parameters, too.

```
Welcome to Bounds 2.1!
```

```
Created by Antti Siirtola 2010-2013 (contact: antti.siirtola@aalto.fi)
```

```
Reducing a parameterised trace refinement task to a finitary one.
```

```
(system: (Sys\LockAct), specification: Spec, valuation formula: ForestTopo)
```

```
Determining a cut-off (size) for the type R...
```

...Succeeded. The cut-off size is 3.

Determining a cut-off (size) for the type U...
...Succeeded. The cut-off size is 2.

Hence, in the case of SRS, it suffices to consider instances of the system and the specification with at most three resources and two users.

If there were no relation symbols, determining the instances up to the cut-offs would be easy. The problem with relation symbols is that not all the values are allowed and there are lots of isomorphic values, which result in the instances of the parameterised verification task with the same answer, too.

Bounds computes the allowed values for parameters one by one. By default, the parameters are treated in the following order:

1. free variables in the alphabetical order,
2. relation symbols in the alphabetical order,
3. types in the alphabetical order.

However, a type T is always handled before variables and relation symbols of the type T. Therefore, the values of some type are always computed first. For example, in the case of our example, the order in which the values of the parameters are computed is 1. R, 2. Anc, 3. U.

As the computation may take a long time, Bounds shows the status of the computation as a percentage value. However, this value does not increase linearly in time, but gives only a rough idea how long the computation will take. Bounds shows also some other statistics that reflect the status of the computation: the total number of valuations computed so far, the number of canonical forms computed and the number of valuations currently stored. The last value converges to the number of instances output.

```
Computing the values of parameters...
    90%   #val:104           #can:20           #sto:13
```

After computing the valuations, the instance generator outputs the corresponding instances of the parameterised verification task. In the case of SRS, there are 14 of them. However, some of them contain redundant information and can be discarded. That is why Bounds analyses each instance in more detail and keeps only those instances which must be refinement checked. Due to parallel computing, the instances are processed in more or less random order. The remaining instances are stored in files named `xxxx_instance_k.csp`, where `xxxx` is the name of the input file without the suffix and `k` is a value from 0 to the number of valuations computed. The syntax of output files is machine-readable CSP.

```
Found 14 non-isomorphic valuations satisfying the valuation formula in 0.03 seconds.
(#valuations: 106, #canonical forms: 20, max #valuations stored: 30)
Generating and analysing Instance 2 generated by valuation
```

```
R -> {R_1,R_2}
U -> {U_1}
Anc -> {}
```

Generating and analysing Instance 1 generated by valuation

```
R -> {R_1}
U -> {U_1,U_2}
Anc -> {}
```

Generating and analysing Instance 0 generated by valuation

```
R -> {R_1}
U -> {U_1}
Anc -> {}
```

```
.
.
.
```

Generating and analysing Instance 11 generated by valuation

```
R -> {R_1,R_2,R_3}
U -> {U_1,U_2}
Anc -> {(R_1,R_2)}
```

Discarding Instance 11. The instance has no behaviour or is already covered by smaller o

Generating and analysing Instance 12 generated by valuation

```
R -> {R_1,R_2,R_3}
U -> {U_1,U_2}
Anc -> {(R_1,R_2),(R_1,R_3)}
```

Discarding Instance 12. The instance has no behaviour or is already covered by smaller o

Instance 0 written successfully to file examples/SRS_instance_0.csp

Instance 1 written successfully to file examples/SRS_instance_1.csp

Instance 3 written successfully to file examples/SRS_instance_2.csp

Instance 5 written successfully to file examples/SRS_instance_4.csp

Instance 9 written successfully to file examples/SRS_instance_3.csp

Instance 13 written successfully to file examples/SRS_instance_5.csp

Generated a total of 6 instances in 0.34 seconds.

```

A total number of instances generated: 6
  (a total number of valuations generated: 166
    a total number of canonical forms computed: 20
    the maximum number of valuations stored all at once: 67)
Total time taken: 0.159942 seconds
  (time taken by input processing: 0.003 seconds
    time taken by the computation of valuations: 0.012 seconds
    time taken by output processing: 0.143 seconds)

```

Run `boundsTraceRefinementChecker examples/SRS` to complete the verification.

3.11 Run the Instance Checker

The instances are verified by running `boundsTraceRefinementChecker` script. The script just calls FDR2 for each instance and reports whether all of them are correct, i.e. the whole parameterised system is correct in the sense of the parameterised specification. Alternatively, an error reported by FDR2 is shown, which implies that the system is incorrect.

```
boundsTraceRefinementChecker examples/SRS
```

This FDR release is for academic teaching and research purposes only. For any other use, please contact Formal Systems (Europe) Ltd at enquiries@fsel.com to obtain a commercial licence.

```
==== Checking instance 0 of examples/SRS ====
```

```
.
.
.
```

```
==== The system is correct with respect to the specification! ====
```

```
real 2m12.463s
user 2m11.472s
sys 0m0.184s
```

The statistics at the end is the output of `time` command. In this case, it takes 2 minutes and 12 seconds to check the 6 instances using FDR2 on a laptop with 4GB of memory and Intel Core i5 processor running Ubuntu Gnu Linux 12.04.

3.12 Optimising the Operation of the Instance Generator

There are two ways a user can affect the performance of the tool: by using as many relation-negated conjuncts in valuation constraints as possible and by changing the parameter order.

Use Relation-Negated Conjuncts in Valuation Constraints You should specify the values of variables and relation symbols using as many conjuncts, where relations occur within an odd number of negations, as possible. Formulae of this kind are called relation-negated. The reason why you should do so is that Bounds searches the allowed values of a relation symbols in the subset of order starting from the smallest possible value, the empty relation. If a value, i.e., a relation, that does not satisfy a relation-negated conjunct is encountered, the search tree can be immediately pruned, because the relation-negated formulae cannot be made true by enlarging the values of relation symbols.

In the case of SRS, there is one such a conjunct, `irrefl`, namely. It is clear that if some relation is not irreflexive, then all the larger relations are not irreflexive either. Hence, the search for the values of `Anc` can be pruned as soon as a non-irreflexive value is encountered. However, it does not hold for other conjuncts, `tran` and `ancprop`, which are not relation-negated.

To improve the computation of valuations, we can try to introduce new relation-negated conjuncts in `ForestTopo` such that the model class of the valuation constraint remains the same. For example, we may add the requirement of asymmetry explicitly in `ForestTopo`.

```
vc asymm = \r1, r2 : ! r1 Anc r2 | ! r2 Anc r1
vc ForestTopo = irrefl & asymm & trans & ancprop
```

This improves the search which appears in the reduced number of valuations generated and stored.

```
A total number of instances generated: 6
  (a total number of valuations generated: 106
   a total number of canonical forms computed: 20
   the maximum number of valuations stored all at once: 30)
Total time taken: 0.166437 seconds
  (time taken by input processing: 0.007 seconds
   time taken by the computation of valuations: 0.012 seconds
   time taken by output processing: 0.146 seconds)
```

Here, the speed-up is not observable but in the case of larger examples the influence of using relation-negated conjuncts may be dramatic.

Change the Parameter Order if Needed The other possibility to improve the search is to try to change the order in which the values of parameters are computed. A user can change the order in pursuance of the definition by giving an insertion value for a parameter. For example,

```
type[1] U
type[2] R
relv[3] Anc
```

results in

```

A total number of instances generated: 6
  (a total number of valuations generated: 185
   a total number of canonical forms computed: 40
   the maximum number of valuations stored all at once: 37)
Total time taken: 0.168442 seconds
  (time taken by input processing: 0.01 seconds
   time taken by the computation of valuations: 0.018 seconds
   time taken by output processing: 0.14 seconds)

```

However, in the case of SRS, the default order is the optimal one.

4 Case: Host Configuration Protocol (HCP)

Next, we consider a host configuration protocol (HCP), where each host repeatedly picks a network address until it finds one that is not used by other hosts. This is done by broadcasting address queries and replies to other hosts in the style of ARP (Address Resolution Protocol). Our goal is to formally model the protocol with an arbitrary number of hosts and an arbitrarily large address space and prove that in our construction, each address is possessed by at most one host.

In HCP, there are two kinds of replicated objects: hosts and addresses. Hence, we use a type T to represent the set of the identifiers of hosts and a type T to denote the set of available addresses. We also introduce some variables which we use to refer to the hosts and addresses.

```

type H
type A

var h : H
var h2: H
var a : A
var a2: A

```

The events used in HCP come in three forms: `ihave(h,a)` denotes that the host h has the address a , `timeout` denotes timeout and `whohas(h,a)` means that the host h wants to know whether someone has the address a .

```

chan timeout
chan whohas : H,A
chan ihave : H,A

```

In order to formalise the specification of HCP, we first capture its behaviour from the viewpoint of two hosts (and all addresses) in an LTS DifAdr . Initially, DifAdr allows the host $h2$ to report having any address but after the host h has picked an address a , the host $h2$ is no longer allowed to report having a . The model of the full specification is obtained by letting h and $h2$ to range over all

pairs of different host identifiers and by composing all the resulting instances of `DifAdr` in parallel. Hence, `DifAdr` allows for each host to report only a single unique address.

```

plts DifAdr =
  lts
    I =    [] a2:ihave(h2,a2) -> I
          [] [] a: ihave(h,a) -> S1(a)
          S1(a) = [] a2:[!a2=a] ihave(h2,a2) -> S1(a)
                  [] ihave(h,a) -> S1(a)
    from I

plts Spec = ||h,h2:[!h=h2] DifAdr

```

Above, the variables `a` and `a2` of the type `A` are used in replicated choices, i.e., structures of the form `S1 = [] a:chan(a) -> S2(a)`. That is why `Bounds` classifies `A` as a *data* type. Respectively, `H` is considered a *process* type, since the variables `h` and `h2` of the type `H` are used in replicated parallel compositions, i.e., structures of the form `||h:P`.

`Bounds` is able to compute cut-offs only when each type is classified either as a data type or a process type, not both. If some type is used both as a data type and a process type, then `Bounds` asks for a user to input a cut-off. In this case, the result of the verification is of course unsound.

The implementation of `HCP` is created in a similar way as the specification. First, we capture it in an `LTS Host` mainly from the viewpoint of a host `h` and secondarily from the viewpoint of a host `h2`. Initially, `Host` just ignores all timeouts and the messages sent by `h2`. The host `h` can also pick an address `a` and broadcast a query in order to see whether the address is already in use. If the other host `h2` says that it has the address or is about to take the same address, then `Host` returns to the initial state. However, if the query timeouts, then the host `h` can either keep it or discard it and return to the initial state. If the host `h` decides to keep the address, then `Host` goes to a state where `h` listens to broadcasts from other hosts and replies when needed. Specifically, if `h2` queries whether the address `a` is in use, then `h` replies that it already has it.

```

plts Host =
  lts
    I =    timeout() -> I
          [] [] a:whohas(h2,a) -> I
          [] [] a:ihave(h2,a) -> I
          [] [] a:whohas(h,a) -> W(a)
    W(a) =  timeout() -> S(a)
            [] timeout() -> I
            [] ihave(h2,a) -> I
            [] whohas(h2,a) -> I
            [] [] a2:[!a2=a] ihave(h2,a2) -> W(a)

```



```

        [] [] a2:[!a2=a] whohas(h2,a2) -> W(a)
S(a) =   timeout() -> S(a)
        [] ihave(h,a) -> S(a)
        [] whohas(h2,a) -> R(a)
        [] [] a2:[!a2=a] whohas(h2,a2) -> S(a)
        [] [] a2:ihave(h2,a2) -> S(a)
R(a) =   ihave(h,a) -> S(a)
from I

```

Again, as we let h and $h2$ to range over all pairs of different host identifiers and compose the instances of `Host` in parallel, we obtain the model of the implementation from the viewpoint of all hosts and addresses.

```

plts Sys = ||h,h2:[!h=h2] Host

```

Finally, we hide the events irrelevant to the specification and ask whether the implementation is a trace refinement of the specification.

```

pset WTEv = ({}_h,a:{timeout(), whohas(h,a)})
trace refinement: verify Sys \ WTEv against Spec

```

In this case, the structural cut-offs computed by the instance generator are two for the number of hosts and 16 for the number of addresses. However, the further analysis performed by the instance generator reveals that in order to prove the HCP implementation correct for any number of hosts and addresses it is sufficient to check six instances. After running the instance checker, we see that all six instances are correct, which implies that also the HCP implementation is correct for any number of hosts and addresses. In other words, each address can be possessed by at most one host. The whole process of verifying HCP took less than a second. The HCP model can be found in the file `examples/hcp.cps`.

5 Case: SRS Interface Verification

So far, we have considered only the verification of safety properties via trace refinement. However, Bounds lends support to two other notions of correctness, too, compatibility and alternating simulation, namely [2]. For this purpose, we need to distinguish between two kinds event, inputs and outputs. By default, the events are treated as outputs and inputs are marked by putting a question mark in front of the channel name. One can also use an exclamation mark to explicitly mark an event as an output.

When using inputs and outputs, we often talk about interface automata [2] instead of LTSs. The use of inputs and outputs affects the parallel composition, too, such that when an input `chan(a1, ..., am)` is synchronised with the output `chan(a1, ..., am)`, the event becomes invisible, `tau`. Consequently, there is no need for explicit hiding when working with interface automata. Actually, the use of hiding is prohibited when checking for compatibility or alternating simulation.

As an example, we consider a simple variant of the shared resource system (SRS) where the resources are not ordered and we do not make a distinction between read and write accesses. Consequently, there is no need for multiple lock modes either. That is why we model the system by using the following events:

`lockreq(u,r)` the user `u` requests for the lock of the resource `r`,
`lockget(u,r)` the lock of the resource `r` is granted to the user `u`,
`lockrel(u,r)` the user `u` releases the lock of the resource `r`,
`lockfree(u,r)` the lock of the resource `r` is successfully released by the user `u`,
`usebeg(u,r)` the user `u` starts accessing the resource `r`, and
`useend(u,r)` the user `u` is finished with accessing the resource `r`.

5.1 Compatibility

Two interfaces are compatible if there is *some* environment process such that when the interfaces and the environment are composed in parallel, it is not possible to reach a state where one interface is willing to execute an output event `chan(a1,...,am)` but the other interface blocks the input event `chan(a1,...,am)`. This is an optimistic view to compatibility since it is sufficient that there is at least one environment where outputs can be consumed immediately.

Our one goal is to model a user, resource and lock (mutex) interface and show that they are compatible. Since the SRS system is modelled in an analogous way as earlier, we do not go into modelling details here but just give the CPS code.

```

type U
type R

var u : U
var u1 : U
var u2 : U
var r : R

chan lockreq : U,R
chan lockget : U,R
chan lockrel : U,R
chan lockfree : U,R
chan usebeg : U,R
chan useend : U,R

plts IM2U =

```

```

lts
  I =    ?lockreq(u1,r) -> LR1
        [] ?lockreq(u2,r) -> LR2
  LR1 =  lockget(u1,r) -> LG1
        [] ?lockreq(u2,r) -> LR12
  LR2 =  lockget(u2,r) -> LG2
        [] ?lockreq(u1,r) -> LR12
  LR12 = lockget(u1,r) -> LG1R
        [] lockget(u2,r) -> LG2R
  LG1 =  ?lockrel(u1,r) -> LF1
        [] ?lockreq(u2,r) -> LG1R
  LG2 =  ?lockrel(u2,r) -> LF2
        [] ?lockreq(u1,r) -> LG2R
  LG1R = ?lockrel(u1,r) -> LF1R
  LG2R = ?lockrel(u2,r) -> LF2R
  LF1 =  lockfree(u1,r) -> I
        [] ?lockreq(u2,r) -> LF1R
  LF2 =  lockfree(u2,r) -> I
        [] ?lockreq(u1,r) -> LF2R
  LF1R = lockfree(u1,r) -> LR2
        [] lockget(u2,r) -> LF1G
  LF2R = lockfree(u2,r) -> LR1
        [] lockget(u1,r) -> LF2G
  LF1G = lockfree(u1,r) -> LG2
        [] ?lockrel(u2,r) -> LF12
  LF2G = lockfree(u2,r) -> LG1
        [] ?lockrel(u1,r) -> LF12
  LF12 = lockfree(u1,r) -> LF2
        [] lockfree(u2,r) -> LF1
from I

plts IMtx = || u1,u2 : [! u1 = u2] IM2U

plts IUR =
  lts
    I =    lockreq(u,r) -> RD
          [] tau() -> I
    RD =   ?lockget(u,r) -> L
    L =    usebeg(u,r) -> WR
          [] lockrel(u,r) -> F
          [] tau() -> L
    WR =   ?useend(u,r) -> L
    F =    ?lockfree(u,r) -> I
  from I

plts IUser = || r : IUR

```

```

plts IR2U =
  lts
    NO =    ?usebeg(u1,r) -> N1
          [] ?usebeg(u2,r) -> N2
    N1 = useend(u1,r) -> NO
    N2 = useend(u2,r) -> NO
  from NO

plts IRU =
  lts
    NO = ?usebeg(u,r) -> N1
    N1 = useend(u,r) -> NO
  from NO

plts IRes = (|| u1, u2 : IR2U) || (|| u : IRU)

plts ISRS = (|| r: IMtx) || (|| u: IUser) || (|| r: IRes)

```

Finally, we state the verification task: we want to know whether the components of **ISRS** are compatible.

```
compatibility: verify ISRS
```

The model is processed by **boundsInstanceGenerator** as earlier. However, this time, the output of the instance generator is processed by **boundsCompatibilityChecker** which calls **FDR2** in order to verify the instances. In this case, all the instances are generated and found to be correct within a second, which implies that the user, resource and lock interfaces are compatible for any number of users and resources. This example is found in the file **examples/miniSRS_comp.cps**.

5.2 Alternating Simulation

Intuitively, a concrete interface (or implementation) is an alternating simulation refinement of (or simply refines, implements or is correct with respect to) an abstract interface (a specification) if the concrete interface does not have stronger input assumptions nor weaker output guarantees than the abstract one.

Our other goal is to show that the resource interface can be safely implemented without concurrency control. For that purpose, we introduce the following CPS code:

```

type U
type R

var u : U
var u1 : U
var u2 : U

```

```

var r : R

chan usebeg : U,R
chan useend : U,R

plts IR2U =
  lts
    NO =    ?usebeg(u1,r) -> N1
          [] ?usebeg(u2,r) -> N2
    N1 = useend(u1,r) -> NO
    N2 = useend(u2,r) -> NO
  from NO

plts IRU =
  lts
    NO = ?usebeg(u,r) -> N1
    N1 =  useend(u,r) -> NO
  from NO

plts IRes = (|| u1, u2 : IR2U) || (|| u : IRU)

plts Res = || u : IRU

alternating simulation: verify Res against IRes

```

Again, the model is processed by `boundsInstanceGenerator` but the output is processed by `boundsAlternatingSimulationChecker`. This program uses FDR2 for computing the parallel compositions and MIO Workbench [3] for establishing alternating simulation. This time, the instance generator is finished within a second but running the instance generator takes roughly six seconds. Most of the time is spent on starting Eclipse under which MIO Workbench runs. This example can be found in the file `examples/res_io.cps`.

You should note that Bounds is not able to compute cut-offs for compatibility and alternating simulation when relation symbols are being used.

6 Further Information

For further information on the theory behind CPS and the operation of Bounds, see my Ph.D. thesis [5] and the conference papers [8, 9, 4]. Of course, you can always contact me personally, too.

References

- [1] Roscoe, A.W.: Understanding Concurrent Systems. Springer (2010)

- [2] De Alfaro, L., Henzinger, T.: Interface automata. *ACM SIGSOFT Software Engineering Notes* **26**(5) (2001) 109–120
- [3] Bauer, S., Mayer, P., Schroeder, A., Hennicker, R.: On weak modal compatibility, refinement, and the MIO Workbench. In: *TACAS '10*, Springer (2010) 175–189
- [4] Siirtola, A., Heljanko, K.: Parametrised compositional verification with multiple process and data types. In: *ACSD '13*, IEEE (2013)
- [5] Siirtola, A.: Algorithmic Multiparameterised Verification of Safety Properties. *Process Algebraic Approach*. PhD thesis, University of Oulu (2010)
- [6] Haustein, M., Härder, T.: Optimizing lock protocols for native XML processing. *Data Knowl. Eng.* **65**(1) (2008) 147–173
- [7] Siirtola, A., Valenta, M.: Verifying parameterized taDOM+ lock managers. In: *SOFSEM '08*. Volume 4910 of LNCS. Springer (2008) 460–472
- [8] Siirtola, A., Kortelainen, J.: Algorithmic verification with multiple and nested parameters. In: *ICFEM '09*. Volume 5885 of LNCS. Springer (2009) 561–580
- [9] Siirtola, A.: Automated multiparameterised verification by cut-offs. In: *ICFEM 2010*. Volume 6447 of LNCS. Springer (2010) 321–337