Parameterized Verification Problem

Processes often have one or more *parameters*, namely quantities or objects that can vary giving different instances of the system.

Almost all systems are built using parameterization at some level, e.g. i in $PHIL_i$. Often even complete systems have implicit or explicit parameters, such as the number of philosophers.

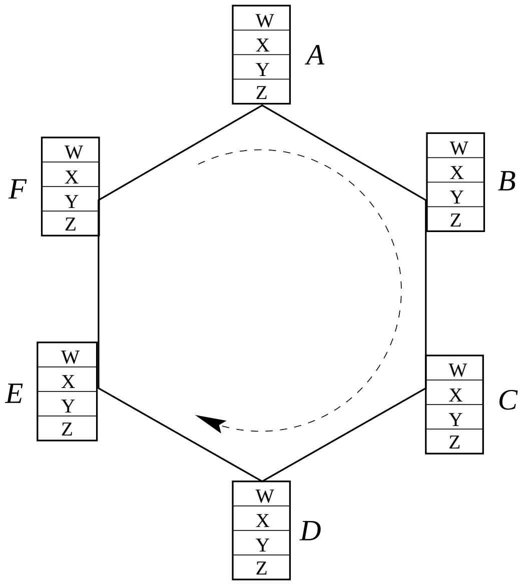
Frequently one has reason to believe that a particular statement is true of a system independently of the value of some parameter: for example, the dining philosophers can deadlock, or are deadlock-free (depending on version) independently of how many of them there are.

Unfortunately FDR can only ever *directly* establish a result for a single parameter value: it requires a pretty concrete system to work with.

The *Parameterized Verification Problem* is the subject of when and how one can establish parameter-independent results about such systems.

Parameters

Consider the ring distributed database:



Parameters here include:

- Number of registers held
- Type(s) of values held in these
- Number of nodes
- Capacity of nodes for messages (travelling and expected-back)
- Ordering used to resolve conflict
- Buffering on channels

Diversity

This richness leads to a wide range of problems and potential solutions, including, for example, induction over the sizes of systems.

Frequently one is reduced to an *ad hoc* argument: for example to reduce many registers in the database to one because updates to distinct ones never affect one another.

The best understood class of parameters, and the one we are going to concentrate on, is that of data types.

We will study the class of *data independent* programs: ones for which some type T is a parameter and *any* nonempty T would make sense.

Criteria for data independence

What we can do with members of T:

Yes

- Input and output them (by themselves, or combined with other values)
- Store them for later use.
- Apply *polymorphic* operations such as tupling and list formation.

No

- Apply any operations that constrain T (e.g., $+$, and) or map from T to any other type (such as comparison operators $<$, $\leq$).
- Use constant values from T
- Compute the size of T and similar
- Indexed constructs (such as $\|$) indexed over T, other than $\prod_{x \in T} P(x)$.

Criteria for data independence II

Maybe

- Compare members of T for equality. *This is the only way our program can ever have its control flow affected by members of T.*
- Have *formal* operations from any type to T (*weak* data independence)

Examples

Data independent

$$
\begin{array}{rcl}
B^\infty(\langle\rangle) & = & \mathit{left}?x \to B^\infty(\langle x\rangle) \\
B^\infty(s\hat{}\langle y\rangle) & = & \mathit{left}?x \to B^\infty(\langle x\rangle\hat{}s\hat{}\langle y\rangle) \\
 & & \Box\ \mathit{right}!y \to B^\infty(s)
\end{array}
$$

$$
\mathit{Pair} = \mathit{left}?x \to \mathit{left}?y \to \mathit{right}!(x, y) \to \mathit{Pair}
$$

Not data independent

$$
\mathit{Max} = \mathit{left}?x \to \mathit{left}?y \to \mathit{right}!(x \lessdot x > y \gtrdot y) \to \mathit{Max}
$$

$$
\mathit{Length} = \mathit{left}?x \to \mathit{right}!(\#x) \to \mathit{Length}
$$

Weakly data independent

$$
\mathit{BinOp} = \mathit{left}?x \to \mathit{left}?y \to \mathit{right}!f(x, y) \to \mathit{BinOp}
$$

(f a function symbol from $T \times T$ to T)

Data independence, type checking and automation

The criteria for data independence are very closely related to those for a type being a *free polymorphic* type in a functional programming language such as ML, Haskell or Miranda. It follows that a decision procedure for whether a type is data independent can be based on a type checker (with one or two customizations and additions).

This is not presently implemented, however it is generally straightforward by hand.

But see later for further remarks on automation....

Parameterized questions we can ask

A data-independent program over type T makes sense whatever non-empty set is substituted for T. If `Spec` and `Impl` are two of them, then it makes sense to ask questions such as

- `Spec` $\sqsubseteq_T$ `Impl`?
- `Spec` $\sqsubseteq_F$ `Impl`?
- `Spec` $\sqsubseteq_{FD}$ `Impl`?
- is `Impl` deterministic?

for any specific T.

In practice one quickly comes to the conclusion that the outcomes of these checks are strongly related. Our objective will be to find *thresholds* beyond which a given check always produces the same answer.

When one exists, it gives an immediate solution to this particular PVP.

Obviously we want to find thresholds that are as small as possible.

Our objective is to show how to calculate them, and to give some of the intuition behind the complex theory that justifies this.

Factors influencing thresholds

The following are the main factors which influence the size (and in some cases the existence) of a threshold for a given refinement check.

- the extent to which the specification constrains which members of T are communicated,
- the kinds of equality checks between members of T that are present in the specification and implementation,
- whether any of the function symbols to T are allowed,
- the subtlety of any nondeterminism in the specification, and
- the extent to which any values of type T which the specification uses are recorded in its traces.

Ranko Lazić has proved many results depending on these factors, falling into four main categories.

When $\mid T \mid = 1$ is enough

If $Impl(T)$ is a program data independent in T, then probably there will be some specification(s) of how it handles T.

But it is also likely that there will be some that make no restriction on what choices it makes of which $x \in T$ to output (and similar). Examples are deadlock and livelock freedom.

If there were equality tests in $Impl$, then even in this simple case the threshold can get arbitrarily high: it might input N values and communicate a iff they are all different. It would pass the specification of never communicating a just when $\mid T \mid < N$.

Let us therefore exclude such tests from both $Spec$ and $Impl$.

NoEqT A data-independent process P satisfies this when it contains *no* test of equality between members of T.

You should bear in mind that there are ways of introducing "hidden" equality tests via parallel communication (synchronizing two outputs of type T) and sets.

Nondeterministic selection

How do we ensure, syntactically, that the specification doesn't care about which member of T is communicated?

By saying that the only way $Spec$ can communicate members of T is with the form `a$x:T -> P` which is introduced as an abbreviation for

```
|~| x:T @ a.x -> P
```

(For the purpose of trace specification, this is equivalent to `a?x:T -> P`.)

For example

```
ALT = a$x:T -> b$x:T -> ALT
```

says that a process (refining it) alternates in communicating members of T along a and b (in appropriate models, without deadlocking or diverging).

Theorem 1

Suppose we have processes $\texttt{Spec}(T)$ and $\texttt{Impl}(T)$, each data-independent (`Impl` perhaps weakly so) with respect to the type parameter T and additionally satisfying condition **NoEqT**. Suppose further that `Spec` is restricted to nondeterministic selections over T as described above.

Then, with $\sqsubseteq$ representing any of $\{\sqsubseteq_T, \sqsubseteq_F, \sqsubseteq_{FD}\}$ the result of the refinement check

$$\texttt{Spec}(T) \sqsubseteq \texttt{Impl}(T)$$

is independent of which non-empty T is used. In particular, the answer for *all* T is revealed by the check when $T = \{0\}$ is a one-element type.

What this says is that if we have a program whose control-flow does not depend on T and a specification which never cares *which* member of T occurs, then the result of the refinement check is independent of T.

So the threshold is 1.

When $\mid T \mid = 2$ is enough

Suppose we have a specification that doesn't fall into the class for Theorem 1, but

- the implementation and specification still satisfy **NoEqT**;
- there are no formal functions to T (we will be banning these *weakly* data independent programs from now on);
- the specification is well structured (in a sense we will develop)

then we will usually be able to have a threshold of 2.

Intuitively, T still doesn't affect $Impl$'s control flow, so all we have to check is that each value gets output in the right place(s). Two values give enough differentiation to do this.

Restriction on specification processes

Things can go wrong with all this if it is possible for a value of type T to appear in the specification without it being unambiguous where it came from.

The clearest way to avoid this and similar difficulties is only to allow immediately apparent, explicit nondeterminism into specifications:

Norm A process `Spec` meets this condition if

(i) its definition contains no hiding or renaming;

(ii) the only parallel operators allowed are alphabetized parallel `P[X||Y]Q` and its replicated version;

(iii) other than the nondeterministic selection construct `a$x:S -> P` no indexed nondeterministic choice is used whose indexing set depends on T;

(iv) all internal and external choice constructs have the initial events of each argument disjoint, and

(v) a few other technical restrictions to ensure the above are not breached implicitly.

Remarks on Norm

Almost all natural *specification* processes one ever builds satisfy this (though it does limit equivalence checking and some subtle specifications such as fault tolerance).

For the conditions of **Norm** are essentially what one would propose as 'clarity' guidelines.

Its name arises because it is effectively designed to ensure that $Spec$ is trivial to *normalize* as required for refinement checking. It means that we can analyze the normal form of the specification and formulate conditions on it easily.

Theorem 2

Suppose $\texttt{Spec}$ and $\texttt{Impl}$ are data-independent processes, both satisfy **NoEqT** and $\texttt{Spec}$ satisfies **Norm**. Let $\sqsubseteq$ be any of $\{\sqsubseteq_T, \sqsubseteq_F, \sqsubseteq_{FD}\}$.

- If $\texttt{Spec}(2) \sqsubseteq \texttt{Impl}(2)$ holds (i.e., for T of size 2) then $\texttt{Spec}(m) \sqsubseteq \texttt{Impl}(m)$ holds for all finite and infinite $m \geq 1$.
- If the refinement $\texttt{Spec}(2) \sqsubseteq \texttt{Impl}(2)$ fails then $\texttt{Spec}(m) \sqsubseteq \texttt{Impl}(m)$ fails for all finite and infinite $m \geq 2$.

Interpretation of Theorem 2

This allows much more powerful specifications that Theorem 1, since they can be specific about which members of T are communicated or refused after any trace. For example $BUFF^N_{\langle\rangle}$ and $WBUFF^N_{\langle\rangle}$ from Chapter 5 are acceptable.

It is frequently possible to cut the state-space considered in the resulting check by forcing all but one of the inputs of type T to be equal. For we can still then check that the one that is different (which could be any of them) comes out at just the right places.

Further details, and more strengthenings, can be found in the papers on data independence referenced in the book.

Theorems 1 and 2 allow you to deal with most practical checks involving data independent systems that do not compare members of T. This covers almost all communication protocols and our distributed database example.

The general case

The implementation

```
RemDups = left?x -> right!x -> RemDups'(x)

RemDups'(x) = left?y ->
              if x==y then RemDups'(x)
              else right!y -> RemDups'(y)
```

falls within our definition of data independence but outside all current theorems. Its control-flow depends on which values of type T it holds.

We can still usually find thresholds for such processes, but they vary with the process much more explicitly. Lazić has shown how to calculate them in terms of coefficients derived from $Spec$ and $Impl$. We will still be assuming that $Spec$ satisfies **Norm**.

Coefficients for threshold calculations

$W^{\texttt{Impl}}$ is the maximum number of values of type T that the implementation `Impl` ever has to store for future use.

$W^{\texttt{Spec}}$ is the corresponding number for the specification.

$L_?^{\texttt{Impl}}$ is the largest number of values of type T that can be input in any *single* visible event of `Impl`.

$L^{\texttt{Spec}}$ is the corresponding number for the specification, except that it additionally takes into account nondeterministic selections (using `$`) which may be present in `Spec`.

$L_{\sqcap}^{\texttt{Impl}}$ is the largest number of values from T that can be nondeterministically chosen in any single nondeterministic choice made over sets involving T in `Impl`. Thus the choice `|~| p:(T,T) @ Q(p)` requires this quantity to be (at least) 2.

All of these quantities *can* be infinite in some circumstances, which would lead (when the coefficient was needed for a threshold) to there being none derivable in that way.

Theorem 3

Suppose `Spec` and `Impl` are data-independent processes, `Spec` satisfies **Norm**, and

$$B \;\geq\; W^{\texttt{Spec}} + W^{\texttt{Impl}} + max(L_{?}^{\texttt{Impl}}, L_{\sqcap}^{\texttt{Impl}})$$

Then for all $N \geq B$, the following traces refinements are equivalent:

$$\texttt{Spec}(B) \sqsubseteq_T \texttt{Impl}(B)$$

$$\texttt{Spec}(N) \sqsubseteq_T \texttt{Impl}(N)$$

Theorem 4

Suppose $\texttt{Spec}$ and $\texttt{Impl}$ are data-independent processes, $\texttt{Spec}$ satisfies **Norm**, and

$$B \;\geq\; W^{\texttt{Spec}} + W^{\texttt{Impl}} + max(L^{\texttt{Spec}}, L_{?}^{\texttt{Impl}}, L_{\sqcap}^{\texttt{Impl}})$$

Then for all $N \geq B$ and $\sqsubseteq$ either of $\sqsubseteq_F$ or $\sqsubseteq_{FD}$, the refinements

$$\texttt{Spec}(B) \sqsubseteq \texttt{Impl}(B)$$

$$\texttt{Spec}(N) \sqsubseteq \texttt{Impl}(N)$$

are equivalent.

The threshold is larger than in Theorem 3 because we have to worry about how members of T get involved in the creation of refusal sets.

Example

```
OnlyDups = left?x -> OnlyDups'(x)

OnlyDups'(x) = left?y ->
                 if x==y then
                 right!y -> OnlyDups'(y)
                 else OnlyDups'(y)
```

Obviously we would expect the process

```
FilterAll = RemDups [right <-> left] OnlyDups
```

(the machine-readable version of `RemDups` $\gg$ `OnlyDups`) not to transmit any value at all. In other words, it should refine

```
Sink = left?x -> Sink
```

Analysis

The various parameters for the refinement check $\mathtt{Sink} \sqsubseteq \mathtt{FilterAll}$ are $W^{\mathtt{Sink}} = 0$, $W^{\mathtt{FilterAll}} = 2$, $L_?^{\mathtt{FilterAll}} = L^{\mathtt{Sink}} = 1$ and $L_\sqcap^{\mathtt{FilterAll}} = 0$. Thus the threshold value for both Theorem 3 and Theorem 4 is 3.

Similarly we could prove that

```
RemDups [|{|left|}|] OnlyDups
```

refines (and therefore is equivalent to) `COPY` with a threshold of 4.

Determinism

Can we compute thresholds which are sufficient to prove that a given data-independent process $\texttt{Impl}(N)$ is deterministic for all larger N? This is important to the checking of security properties.

As with refinement checking, there are a number of results depending on the complexity of the use of T within `Impl`. There are similarities and differences with the refinement results.

We need one more coefficient before we can give the theorems.

$L^{\texttt{Impl}}_{Occur}$ is the largest number of distinct values of type T that can appear in any *single* visible event of `Impl`.

Theorem 5

Suppose `Impl` is data-independent and satisfies **NoEqT**. Then

(a) If $\mathtt{Impl}(L_{Occur}^{\mathtt{Impl}} + 1)$ is deterministic, then $\mathtt{Impl}(N)$ is deterministic for all $N \geq 1$.

(b) If $\mathtt{Impl}(L_{Occur}^{\mathtt{Impl}} + 1)$ is nondeterministic, then $\mathtt{Impl}(N)$ is nondeterministic for all $N \geq L_{Occur}^{\mathtt{Impl}} + 1$.

Note that this result corresponds to the $\mid T \mid = 2$ case for refinement, but that the threshold is occasionally bigger than 2!

Theorem 6

Suppose Impl is data-independent and contains no renaming which conceals the value of any member of T input in a communication, then the threshold for determinism is given by

$$2 \times W^{\texttt{Impl}} + \max(L_{?}^{\texttt{Impl}}, L_{\sqcap}^{\texttt{Impl}})$$

in the sense that $\texttt{Impl}(N)$ is deterministic if and only if $\texttt{Impl}(M)$ is, for M, N at least this value.

This corresponds to Theorem 4. In the case of determinism proper (over $\mathcal{N}$ as opposed to $\mathcal{F}$) it is conjectured that the factor of 2 can be removed.

Underlying theory

The theory of data independence depends on our being able to build relationships between behaviours that occur with different values for members of T.

We need to show that any counter-example for arbitrary type would still be possible for a type of the threshold size.

Lazić has used two distinct methods for this:

- *Parametricity* or *logical relations:* One shows that any relation (perhaps from some restricted class) between two types lifts naturally to a relation between processes. This is borrowed from functional programming, and is the main tool in the presence of **NoEqT**.
- *Symbolic operational semantics:* a transition system is built in which nodes and actions contain variables of type T. It (and its size) are independent of T, since it contains T as a parameter.

Potential automation

- Condition checking (data independence, **Norm**, etc.)
- Decisions about which result applies.
- Constant calculation ($W^{\mathtt{Impl}}$ etc) either syntactically or (for tighter bounds) operationally.
- Symbolic normalization (to allow, usually, the removal of the **Norm** condition).
- Symbolic execution of refinement checks (which should often be more efficient than doing the same checks at the threshold types).