Computing Parameterized Helpful Assertions and Ranking Functions

In this lecture we will present some algorithms for the computation of helpful assertions and ranking for the case of parameterized systems. As in the case of algorithms for computing invariants, these algorithms are based on the analysis of small instances of the parameterized systems.

First, we consider the deductive analysis of finite-state systems, where most of the examples are instances of parameterized systems.

We start by presenting a version of rule WELL which is based on the justice requirements rather than on transitions.

Identifying the Requirement with Minimal Rank

Assume that \( J_{\text{min}} \) is the requirement with minimal positive ranking.

Examining the premises of rule WELL for the case of \( J_{\text{min}} \), we find that assertion \( h_{\text{min}} \) should satisfy:

\[
\begin{align*}
W2. & \quad h_{\text{min}} \land \rho & \Rightarrow & \quad h'_{\text{min}} \lor q' \\
W3. & \quad h_{\text{min}} & \Rightarrow & \quad \lnot J_{\text{min}}
\end{align*}
\]

Namely, we should pick a requirement \( J_{\text{min}} \) whose associated transition can lead in one step to the goal \( q \). We should then determine an assertion \( h_{\text{min}} \) such that \( J_{\text{min}} \) is false on all \( h_{\text{min}} \)-states (W3), and all \( p \)-successors of \( h_{\text{min}} \)-states satisfy \( h_{\text{min}} \lor q \).

Furthermore, \( h_{\text{min}} \) should consist of pending states. These are the states which can be reached from a reachable \((p \land \lnot q)\)-state by a \( q \)-free path. These states can be computed symbolically by the expression:

\[
\text{pend}: (\text{reachable}_p \land p \land \lnot q) \lor (p \land \lnot q)^*
\]
**An Algorithm for Computing Auxiliary WELL Constructs**

**Algorithm** `compute-help` — Compute Auxiliary Constructs for Rule WELL. The non-disjoint case

\[
pend := (reachable_D \land p \land \neg q) \lor \langle p \land \neg q' \rangle^* \\
\]

\[
r := 0 \\
\]

\[
nj := nJ(1) \\
\]

**Fix** (`pend`)

\[
\text{for } i = 1, \ldots, nj \text{ do} \\
\]

\[
\begin{aligned}
& h_{\text{min}} := \forall h : pend \land \neg J[i] \land \text{pre}(\rho, h \lor \neg pend) \\
\text{if } h_{\text{min}} \neq 0 \text{ then } \\
& r := r + 1 \quad \text{— Increment rank} \\
& h[i] := h_{\text{min}} \quad \text{— Set helpful assertion} \\
& d[i] := r \quad \text{— Set rank} \\
& \text{indJ}[i] := i \quad \text{— index of justice requirement} \\
& pend := pend \land \neg h_{\text{min}} \quad \text{— Remove } h_{\text{min}} \text{-states from } pend \\
\end{aligned}
\]

The algorithm places in \( h[1], h[2], \ldots, d[1], d[2], \ldots \) and \( \text{indJ}[1], \text{indJ}[2], \ldots \) the helpful assertions, ranks, and indices of justice requirements, respectively.

**The Case of Disjoint Justice Requirements**

In many cases, it is possible to find lists \( J_1, \ldots, J_n \) and corresponding \( h_{1}, \ldots, h_{nj} \), such that the \( J_i \)’s are pairwise disjoint. To find the \( h_i \)’s for such a case we start by employing an algorithm similar to `compute-help` except that its innermost block is replaced by

\[
\begin{aligned}
\text{if } h_{\text{min}} \neq 0 \text{ then } \\
& h[i] := h[i] \lor h_{\text{min}} \quad \text{— Add helpful assertion} \\
& pend := pend \land \neg h_{\text{min}} \quad \text{— Remove } h_{\text{min}} \text{-states from } pend \\
\end{aligned}
\]

where, \( h[1], \ldots, h[nj] \) are initially set to 0. Thus, it is assumed that, while computing a new candidate for \( h_{\text{min}} \), we may encounter a candidate corresponding to \( J[i] \) more than once. This version takes as \( h[i] \) the disjunction of all candidates corresponding to \( J[i] \).

This part of the modified algorithm does not compute the ranks, which are no longer given by the index of the assertion/justice requirement.

In order to compute the ranks we employ an additional algorithm which performs topological sort on the assertions. The lowest rank is assigned to an \( h[i] \) which has no successor in any \( h[j], j \neq i \). Once such an assertion is identified, it is removed from the list, and the process is repeated.

**An Example in TLV**

To apply this process to program \( \text{TOKEN-RING}(5) \), we place in file `tokenr-help.pf` the following script:

To `compute_inv`:

- `Let inv := 1;`
- `For (i in 1...N)`
- `Let inv := inv & (P[i].loc in {1,4,5} -> k=i);`

End -- `For (i in 1...N)`

End -- `To compute_inv;`

`compute_inv;`

Call `binv(inv);`

Load "compute-constructs.pf";

Print "\nCompute Constructs for stringn";

`compute_help P[N].loc=3, P[N].loc=4;`

Call `wellx-nond(P[N].loc=3, P[N].loc=4, inv, rank, h, d, indJ);`

where `wellx-nond` is the procedure that checks the premises of rule WELL for the case of non-disjoint justice requirements.
Example in TLV

To apply the method of disjoint justice requirements, we place in file `tring-nond.pf` the following script:

To compute-inv;
Let inv := 1;
For (i in 1...N)
  Let inv := inv & (P[i].loc in \{1,4,5\} \rightarrow k=i);
End -- For (i in 1...N)
End -- To compute-inv;
compute-inv;
Call binv(inv);

where `compute-construct` is the procedure that produces helpful assertions with unique associated justice requirements, `rank-by-sort` computes ranks by topological sorting, and `wellex` is a procedure that checks the premises of rule WELL for the disjoint-requirements case.

Computing Distributed Ranks

Often, we may wish to use rule DISTR-RANK for proving a response property. Recall that a good insight for designing the distributed ranks \( \delta[i] \) is to take them as assertions (i.e., functions ranking over \( \{0,1\} \)), where state \( s \) satisfies \( \delta[i] \) iff there exists a pending path from \( s \) to a state \( s' \) that satisfies \( h[i] \). Consequently, we use the following algorithm for computing distributed ranks:

Algorithm compute-distr — Compute distributed ranks

\[
\text{for } i = 1, \ldots, nj \text{ do } \delta[i] := (\text{pend } \wedge \rho)^* h[i]
\]

Uniform Verification of Parameterized Systems

All the examples shown above were of individual instances of parameterized systems. We will now consider methods for uniform verification of progress properties of parameterized systems. These methods establish in one application that \( S(N) \models p \Rightarrow \Diamond q \), for every \( N \).

The method is based on the project & generalize approach which consists of the following steps, where we assume that we intend to apply rule DISTR-RANK with helpful assertions and boolean rankings of the form \( h_j[i] \) and \( \delta_j[i] \), where \( i \in \{1..N\} \) and \( j \) ranges over a fixed domain \( j \in \{1..k\} \), independent of \( N \):

- Choose an appropriate cutoff value \( N_0 \).
- Apply algorithms compute-constructs and compute-distr to \( S(N_0) \). As a sanity check, establish that \( S(N_0) \models p \Rightarrow \Diamond q \).
- Choose a generic index, say \( i = 3 \). Generalize \( h_j[3] \) and \( \delta_j[3] \) to \( h_j[i] \) and \( \delta_j[i] \), respectively, for all \( i \in \{1..N_0\} \) and \( j \in \{1..k\} \).
- Invoke rule DISTR-RANK with the generalized versions of \( h_j[i] \) and \( \delta_j[i] \).