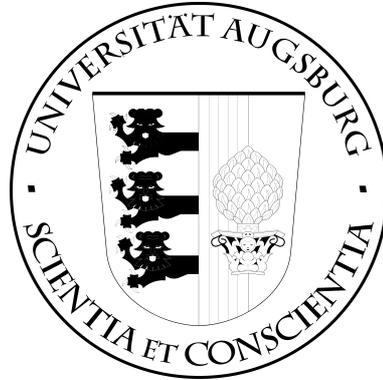


UNIVERSITÄT AUGSBURG

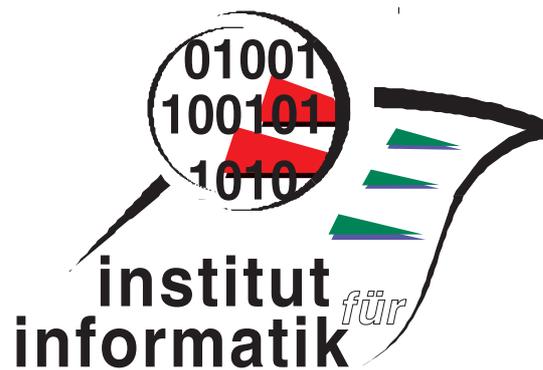


**DesiJ – A Tool for STG Decomposition**

**Mark Schaefer**

Report 2007-11

October 2007



**INSTITUT FÜR INFORMATIK**

D-86135 AUGSBURG

Copyright © Mark Schaefer  
Institut für Informatik  
Universität Augsburg  
D-86135 Augsburg, Germany  
<http://www.Informatik.Uni-Augsburg.DE>  
— all rights reserved —

# DESIJ – A Tool for STG Decomposition

Mark Schaefer\*

Institute of Computer Science, University of Augsburg, Germany  
mark.schaefer@informatik.uni-augsburg.de

## Abstract

For synthesising efficient asynchronous circuits from STGs one has to deal with the state space explosion problem. For this purpose STG decomposition was introduced by Chu and improved by Vogler and Wollowski.

Here, we present the tool DESIJ which is the implementation of STG decomposition and proposed by Vogler and several other optimisations.

**Keywords:** Asynchronous circuit, STG, Petri net, decomposition, state space explosion, DesiJ.

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>STG Undo Implementation</b>	<b>3</b>
<b>3</b>	<b>Calculation of Decomposition Trees</b>	<b>7</b>
<b>4</b>	<b>Conditions, Collectors and Operations</b>	<b>8</b>
<b>5</b>	<b>Verification of the Implementation</b>	<b>9</b>
<b>6</b>	<b>Command Line Options and Parameters</b>	<b>11</b>
6.1	General Options and Parameters . . . . .	11
6.2	Control of the Decomposition Algorithm . . . . .	12
6.3	Handling of Redundant Places . . . . .	14
6.4	Synthesis of Components . . . . .	14
6.5	Various Options . . . . .	15
6.6	Output Options . . . . .	15

---

\*This research was supported by DFG-projects 'STG-Dekomposition' Vo615/7-1 and Wo814/1-1.

*Premature optimization is the root of all evil  
– or at least most of it – in programming.*

*Donald Knuth, 1974*

## 1 Introduction

In this chapter, the implementation DESIJ (decomposition Java) of the decomposition algorithm is described. Based on [VW02, VK05], B. Kangsah implemented a prototype version DESI in C, which covered the basic decomposition algorithm. Since this implementation was simply not extendable, we decided to start a new implementation from scratch. This time Java was chosen instead of C++ (or even C as for DESI), in order to focus on the implementation itself instead of technical details of the programming language, as this often happens for C/C++. Furthermore, performance was not the prime objective for the following reasons:

- The decomposition algorithm works only structurally, i.e. it is explicitly avoided to perform expensive operations in particular building reachability graphs.
- The main focus was the easy extensibility with the new features developed during the STG decomposition project.
- Additionally, the performance of Java is not so bad compared to C/C++ anymore, due to the availability of Java virtual machines with just-in-time-compiling.

To guarantee the extensibility, DESIJ was developed in three layers, based on the Java runtime environment (JRE) and some external libraries, see Figure 1:

- The STG layer provides basic functionality for STG handling independent of decomposition.
- The decomposition layer implements the various decomposition strategies [SVWK06] and provides features for the generation and handling of handshake circuits as STGs.
- The interface layer provides three access possibilities for the implemented functionality: a batch command line mode, which is the most powerful mode with the most options, an interactive command line mode and a graphical user interface for decomposition. The features of the command line mode are introduced in Section 6.
- The Condition, Collectors and Operations block is something special: these operations are independent of decomposition but provide useful features for it which belong to the STG layer on a functional level, nevertheless. They are described in Section 4.
- Additionally, there are independent blocks with prototype implementations of new ideas and concepts (not shown).

This report is organised as follows: in the first section, some implementation issues are discussed and justified. In Section 6 an overview of the implemented functionality of DESIJ is given in form of its commented command line options.

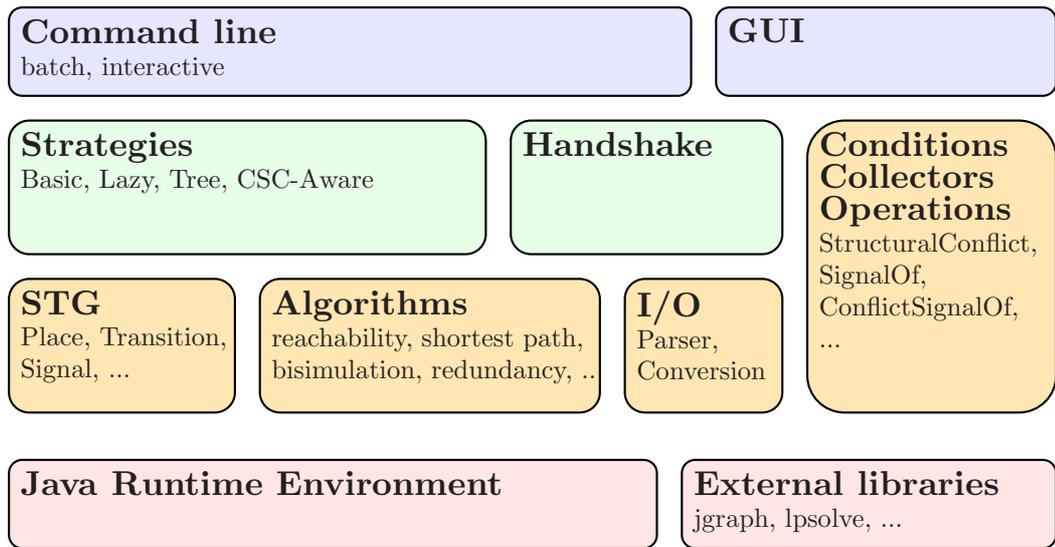


Figure 1: DESIJ Structure

## 2 STG Undo Implementation

In the first implementation of the decomposition algorithm, backtracking was realised by generating copies of the respective component STGs. This is inefficient for `LAZYBACK`, `TREE` and `AGGREGATION`, because between two savepoints, decomposition-tree nodes resp. often only small parts of a component are modified. Undoing this few changes is clearly more efficient than to copy all the unchanged parts, especially for a large STG. In `BASIC` and `REORDERING`, backtracking means to restore the initial component. If this performed for a small intermediate STG, copying might be more efficient: in particular, during reduction a lot of redundant places are produced and deleted afterwards. When undoing, the places are first introduced and then deleted again. Nevertheless, it turned out that even these strategies benefit from undoing.

Furthermore, copying the STGs increases the memory usage very much, because there are a lot of intermediate results which have to be stored – this is especially important for `TREE` where all components are created at the same time and even more intermediate results have to be stored.

Instead, in DESIJ an undo mechanism is implemented which can restore previous versions of an STG very efficiently. Functionally, it works as a stack like in most applications (e.g. text or image editors), i.e. each time the last change is undone.

There are only three supported operations: adding/removing nodes with their incident arcs and changing the signature. All high-level operations like transition contraction are reduced to these ones; at the moment, there is no operation for changing arc weights, since it is not needed.

For every change in an STG the corresponding undo operation (`UndoOperation`) is generated and pushed on the undo stack (`STG.undoStack`). When an undo is performed, the last operation is removed from the undo stack and is applied, i.e. the corresponding change is undone. (`UndoOperation.apply()`).

To understand the unod mechanism, we will first have a short look on the data structure of an STG in DESIJ, cf. also the first row in Figure 2. Essentially an STG is a graph (and the undo

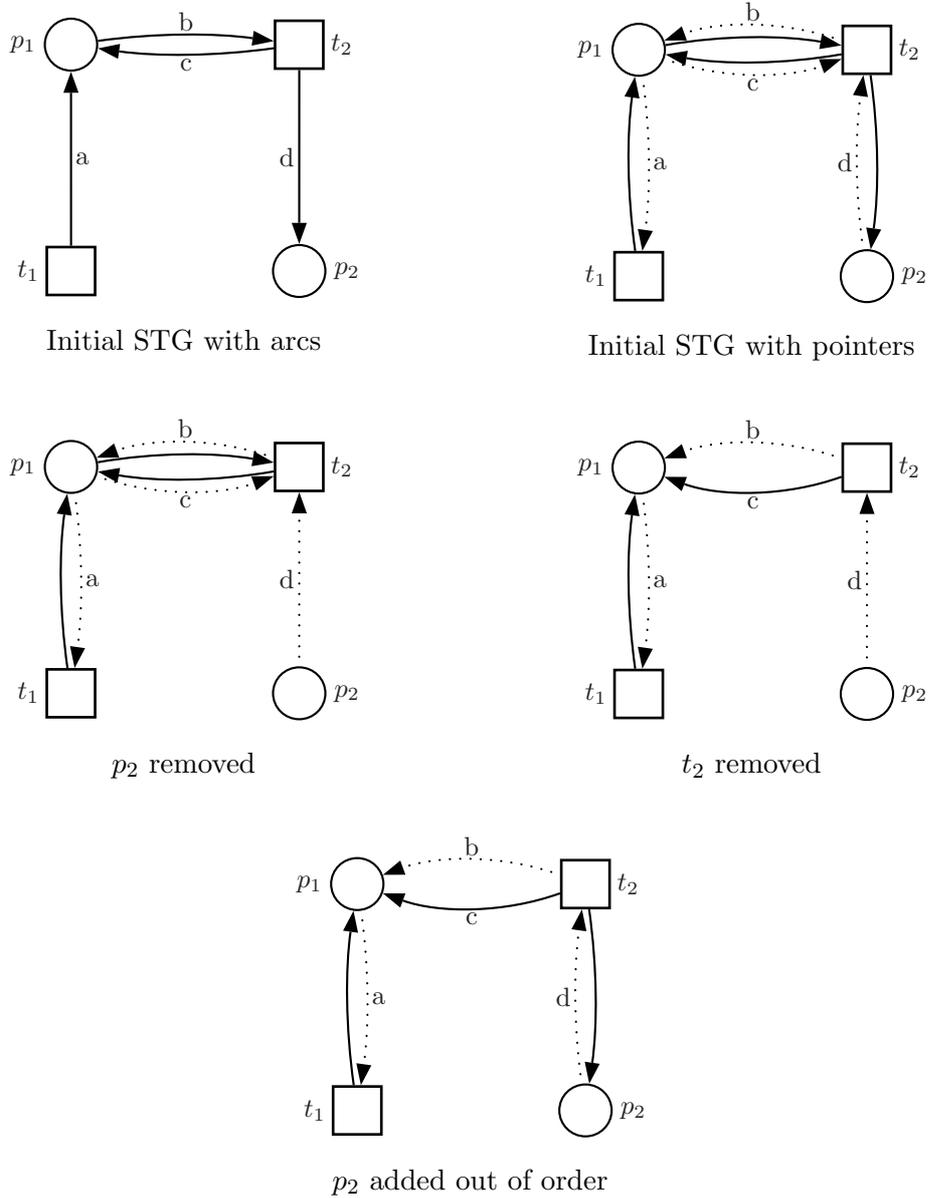


Figure 2: STG undo example. The internal representation of the initial STG (upper left) contains two pointer for every arc (upper right) – a forward pointer and a backward one (dotted). Here, the arc labels denote the arc names.

stack can also applied to these), and it is stored with adjacency lists kept in the nodes. The nodes are double-linked, i.e. for every arc from node  $x$  to node  $y$  of the net, two pointers are stored: one *forward pointer* in  $x$  pointing to  $y$ , and one *backward pointer* in  $y$  pointing to  $x$ .<sup>1</sup> Furthermore, all nodes of a net are stored in a list.

Undoing the adding of a node is quite easy: the respective node and incident arcs are simply removed from the STG. Undoing the removal of a node  $x$  is a bit more complicated and harnesses that the nodes are double-linked: if  $x$  should be removed from a net, it is not deleted completely, but only removed from the list of nodes and all pointers pointing to  $x$  are deleted in the respective nodes, i.e. the STG ‘forgets’ about  $x$ . On the other hand, the outgoing pointers of  $x$  are kept, i.e.  $x$  itself still ‘knows’ about its former connections.

It depends on the direction of an arc which of the corresponding pointers is deleted, e.g. in the STG in Figure 2,  $t_2$  lies on a loop with  $p_1$  formed by the arcs  $b$  and  $c$ . When  $t_2$  is deleted, the backward pointer of  $c$  but the forward pointer of  $b$  is deleted in  $p_1$ . If the deletion of  $x$  should be undone, all ingoing pointers to  $x$  are restored in the respective nodes from the outgoing counterparts in  $x$ , and  $x$  is added again to the list of nodes.

For an example, have a look at Figure 2. The initial STG (upper left) contains four nodes and four arcs ( $a$ ,  $b$ ,  $c$  and  $d$ ). In the internal representation (upper right), each of these arcs is represented by a forward pointer and a backward pointer (dotted). If the nodes  $p_2$  and  $t_2$  are removed in this order, we get the internal representations in the middle row. As one can see, only the respective outgoing pointers remain. If the operations are undone, the states are encountered in the reverse order.

In principle, it is possible to apply undo operations out of order as in the last row, where  $p_2$  was added before  $t_2$ . Observe, that this STG is not well-formed: the backward pointer of arc  $d$  is *dangling*, i.e. it points to a node which is not contained in the list of nodes. Therefore, only STGs corresponding to a proper stack state<sup>2</sup> are guarantee to be well-formed, but it is also possible that independent operations are undone, resulting in well-formed data structures, e.g. if the respective nodes are not adjacent. Undoing out-of-order might allow to perform undo operations concurrently on multi-processor systems.

The undo operations for the modification of the signature just keep track of the old state and restore it if needed. For example, the STGs keeps record of each occurring signal (also for lambda-rised signals, the signal name is kept – the signal type is just changed to dummy). If the last transition of a signal is deleted, this signal is removed from the list of signals and re-inserted if the deletion of the transition is undone.

Additional features of the undo stack are combined undo operations which encapsulate associated modifications; at the moment, this is only used for transition contractions. Furthermore, one can push special parametrised *undo markers* onto the stack. They are used to simplify backtracking: the BASIC strategy puts a marker on the stack before the reduction starts; if it is necessary to backtrack, the `STG.undoToMarker(Object)` method is called to undo all operations performed after the respective marker was added. Different strategies use different markers, which makes it

<sup>1</sup>Java does not have pointers in the same sense as C++ has, but all variables in Java are actually references and therefore some kind of pointer, although there is no pointer arithmetic.

<sup>2</sup>We can imagine that non-top-elements removed from the stack are replaced by dummy elements, which will be removed automatically if they become the top element. Then, a stack is proper if it contains no dummy elements, and this is exactly the case if this state was encountered previously when applying the operations which are now undone.

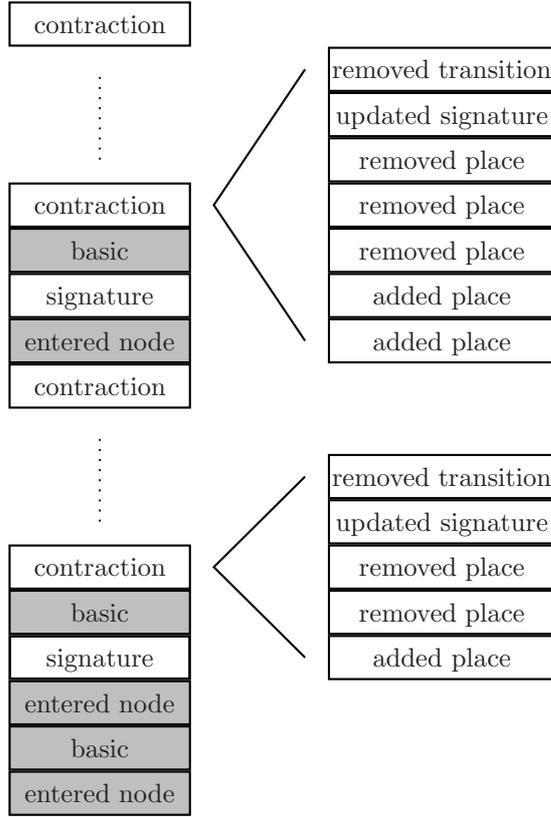


Figure 3: Undo stack during TREE. Gray entries are undo markers, contractions are combined undo operations consisting of several operations.

possible that the other strategies can use BASIC to perform reduction in an intermediate STG: BASIC only undoes its own changes when backtracking, and other strategies can undo to their own markers ignoring the BASIC marker.

A typical stack for TREE might look as in Figure 3 (undo markers are filled). At the bottom of the stack, there is a ‘entered node’ marker, it corresponds to the begin of the decomposition algorithm when the root node of the decomposition tree is entered with the specification STG. Then BASIC is performed for the signals to be contracted in this node. A ‘basic marker’ is set to enable backtracking here; as one can see, directly on top of this marker there is another ‘entered node’ marker, i.e. in the root node no contractions took place. Then, the first real change happened, viz. the type of some signals was changed to dummy (i.e. they were lambda-rised), and the contractions of the corresponding transitions took place (again preceded by a basic marker). Each undo operation encapsulating a contraction is built from several smaller undo operations as described above: first the new combined places are added, then the old ones are removed, then the signature is updated (it is possible that the transition was the last of a signal<sup>3</sup>), and finally the contracted transition is removed. In the lower contraction, the respective transition has exactly one place in its pre- and postset; in the upper contraction, either the preset or the postset contains two places.

<sup>3</sup>The STG implementation does not forget about lambda-rised signals: the former signature of such signals is just replaced with a special dummy signature. Signals without corresponding transitions are removed, however.

### 3 Calculation of Decomposition Trees

For the decomposition strategy TREE [SVWK06], it is needed to precalculate a *decomposition tree* which is the plan for the decomposition process. Every node stores a set of signals which should be lambda-rised and contracted when entering it with an STG, and decomposition starts at the root node with the initial STG. Therefore, every leaf corresponds to a final component, in which all signals on the path from the root to this leaf are contracted.

Since the purpose of this tree is to minimise the absolute number of contractions while generating all components, we consider a tree as optimal if the overall number of signals in the nodes is minimal. This is a slightly relaxed assumption, since the number of transitions of a signal is not constant, but in practical STGs most signals have only two transitions. Furthermore, as it was mentioned in [SVWK06], the tree will be changed during decomposition due to postponing contractions.

In general calculating optimal decomposition trees is NP-complete (see [KK01], decomposition trees are called preset trees there). Also in [KK01], a heuristic algorithm is given, which performs reasonably well: it starts with a set of trees – one for each component, containing only one node with all signals to be contracted in the respective component. Then the algorithm works bottom-up, combining root nodes which share the most signals, until only one tree remains.

For these calculations, a data structure which is a combination of an associative array and a sorted tree is used for storing all pairs of root nodes together with the intersection of their signal sets. It supports the following operations ( $n$  is the number of signals):

- Given two sets of signals, adding the entry for the intersection of them is done in  $O(n)$ :  $O(n)$  for generating the intersection and  $O(\log n)$  for adding.
- Retrieving the largest intersection is done in  $O(1)$ .
- Given two sets of signals, removing all corresponding entries and adding new entries for their intersection is done in  $O(n^2)$ .

Initially, this data structure contains all possible intersections between the signals of all root nodes. If  $n$  is the number of signals of the specification, setting this up takes  $O(n^3)$  time,  $O(n^2)$  entries of  $O(n)$  size. Obviously, also the initial memory usage is in  $O(n^3)$ . If one wants to generate the finest partition or, in general, decompositions with STG components of a constant size, i.e. not depending on the size of the specification, the memory usage as well as the runtime is in  $\Omega(n^3)$ .

The main loop iteratively retrieves the largest intersection, combines the corresponding trees and updates the data structure. The trees are combined by generating a new tree node which stores the intersection and which has these trees as children. Furthermore, the intersection is removed from the signal set stored in these trees. Since in every step the number of trees is decreased by 1, the main loop is executed exactly  $n - 1$  times resulting in a  $O(n^3)$  overall runtime.

To make the implementation more efficient, the signal sets are stored as fixed length bitsets, which improves the runtime and memory usage by a constant factor. If this is still not efficient enough, the algorithm can be partially randomised: the set of leaves is (randomly) partitioned into sets of a fixed size and the presented bottom-up algorithm is applied to each of them separately, resulting in a number of preset trees. The root nodes of these trees are considered as leaves for a new iteration and so on until only one tree remains.

For a large STG, the decomposition tree can now be calculated in less than a minute; this tree reduces the number of contractions to about 15% compared to the BASIC strategy. In random benchmarks which mimic the characteristics of STG decomposition trees<sup>4</sup>, the bottom up algorithm is about 2 percentage points better than the partially randomised version. The completely random strategy, which combines in each step two arbitrary trees, is of course even more faster – runtime and memory usage are in  $O(n^2)$  – and reduces the number of contractions to about 20%. This algorithm will possibly be needed for larger specifications.

## 4 Conditions, Collectors and Operations

The condition/collector/operation concept was introduced to enable the fast (sometimes prototype) implementation of the necessary algorithms.

Conditions (`stg.traversal.Condition`) check if a given object fulfils some properties, for instance if the signal of a transition is in a given set, or if a node is the child of another node. Furthermore, there are special classes which allow to combine predefined conditions to more complex conditions: the `NotCondition` just negates a condition, the class `MultiCondition` combines a set of conditions with AND, OR or XOR.

Collectors (`stg.traversal.Collector`) derive certain information from objects, e.g. the children or syntactical triggers of a node. An overview of the currently implemented conditions and collectors can be found in Tables 1 and 2. Additionally, operations (`stg.traversal.Operation`) can perform certain tasks on given elements; there are no predefined classes.

Although each condition or collector could be used stand-alone or for just one object, they are usually used in combination with one of the following methods. `List<T>` denotes a list with elements of Type T, `Condition<T>` denotes a condition which can be applied to objects of type T and `Collector<T,R>` denotes an operation which is applied to objects of type T and returns objects of type R. A `Collection` is the most general superclass (except the `Object` class itself) of the `List` class; it defines the basic interface for representing a bunch of objects.

- `List<T> getElements(Collection<T> c, Condition<T> cond)`  
Returns the elements from `c` for which `cond` is true.
- `List<R> collect(Collection<T> c, Condition<T> cond, Collector<T,R> col)`  
Returns the results of `cond` for the elements of `c` for which `cond` is true.
- `void modify(Collection<T> c, Condition<T> cond, Operation<T> op)`  
Applies the operation `op` to all elements of `c` for which `cond` is true.

In some cases the usage of a condition or collector was replaced with a specialised implementation, either for efficiency or due to growing complexity. In particular, the `Contractable` condition was moved to `STG.isContractable(STG stg, Transition transition)`: checking if a contraction is applicable has become quite complex due to the development of several optional conditions and the need to analyse the exact reason during decomposition, if the contraction is not possible.

---

<sup>4</sup>The size of the initial sets as well as their number is related to the number of signals as it is the case for STGs, e.g. each initial set contains nearly all signals.

In general, during the development of DESIJ, the condition/collector/operation concept turned out to be very useful. It allows to implement pseudo-code style descriptions of an algorithm in a very straightforward way, because it separates different layers of implementation from each other.

## 5 Verification of the Implementation

In most cases, scientific programming is inherently more complex than business programming: there are no predefined data structures and the algorithms are not well-known but newly developed.

It is therefore even more important to ensure the correct functionality of the developed software. In particular, DESIJ works on large input files which cannot be checked by hand. While this is possible for small benchmarks, it is easily possible that new effects and errors arise only for large STGs.

To guarantee<sup>5</sup> the correctness of the decomposition algorithms, the following approach was chosen: an independent package of DESIJ tries to find a proper STG bisimulation for an STG and its decomposition. If this is always successful, even for large STGs, the implementation of the corresponding STG decomposition algorithm is considered correct. This introduces redundancy: since the two parts of DESIJ only share the low level implementations of the STG class, it is rather unlikely that both parts work incorrect in a way such that a correct decomposition is reported.

The correctness checking package is divided into two parts:

- A general search engine (`Simulation`) which tries to build a relation between the states of two systems which satisfies certain rules.
- A set of `RelationPropagators` which define the corresponding rules. Rules are of the form: if this element is in the relation, then also this *or* this *or* ... element has to be. A rule also defines which elements have to be in the relation unconditionally.

In particular the `STGBisimulationPropagator` defines the rules of an STG-bisimulation.

---

<sup>5</sup>Of course, this is no 100% guarantee in the sense of verification. The implemented algorithms itself are proven correct anyway – this is a check of their correct implementation.

Name	Description	Comb.
Activated	If a transition is activated	
Adjacent(Set<Node> n)	If a node is adjacent to a node from n	•
All	Always true	
ArcWeight(int n)	If a node is incident to arcs with weight > n	
ChildOf(Node n)	If a node is a child of n	•
Contractable	If a transition can be contracted. Not used any more, cf. text.	•
DuplicateTransition (Transition t)	If a transition is a duplicate of t	
EqualTo(Object o)	If an object equals o (in the sense of Java)	
LoopOnly	If a node is a loop-only node	
LoopWith(Node n)	If a node forms a loop with node n	•
MarkedGraphPlace	If place has only one incoming and one outgoing arc, both with weight 1	
MultiNodes(Set<Node> n)	If a node is contained in n	
NewAutoConflict	If a transition contraction creates a new auto-conflict pair	
NumberOfChildren(int n)	If a node has more than n children	
NumberOfParents(int n)	If a node has more than n parents	
ParentOf(Node n)	If a node is a parent of n	•
RedundantPlace	If a place is redundant. The most complex condition.	•
RedundantTransition	If a transition is redundant	
SafeContractable	If a contraction preserves safeness structurally	
SecureContraction	If a contraction is secure	•
SelfTriggeringPlace	If a place connects two transitions labelled with the same signal	•
signal/signature	Several conditions related to the labelling of transitions	
StructuralConflict (Transition t)	If a transition is in structural conflict with t	

Table 1: Implemented Conditions. The ‘Combined’ column is marked if the respective condition is a `MultiCondition` or `NotCondition`, or if its implementation mainly uses other conditions. The `All` condition may seem a bit pointless, it is however used if `collect` or `modify` should be applied without any preconditions.

Name	Description
AutoConflict	Returns the signals for which a place constitutes a structural auto-conflict
Children	Returns the children of a node
ConflictSignal	Returns all signals which are in structural conflict for a transition
Identity	Returns the object itself
Marking	Returns the marking of a place
NewAutoConflictPair	Returns all signals for which a contraction will create new structural auto-conflicts
Parents	Returns the parents of a node
Signal	Returns the signal of a transition
String	Returns a string representation of an object
SignalTrigger	Returns all weak syntactical triggers of a transition
SyntacticalTrigger	Returns all syntactical triggers of a transition

Table 2: Implemented Collectors.

## 6 Command Line Options and Parameters

In this section, the various command line options of DESIJ are listed, in order to give an overview of the implemented features and to give an impression of the complexity of the whole system.

There are command line options and command line parameters. The first ones can be given in a long and in a short form. For instance, the help option can be given as `--help` or just as `-h`. The short forms can be combined, i.e. instead of `-m -Z`, just `-mZ` can be used. Furthermore, options have a default value, either true or false, which is switched by the presence of the option. The default value is given in brackets on the right. Parameters are given in the form `parameter=value`. If there is a default value, it is also given in brackets.

The default values for options and parameters are chosen such that in most cases just typing `desij some_stg` results in a fast and sufficient decomposition of `some_stg`.

### 6.1 General Options and Parameters

<code>--help</code> Shows the help message.	<code>-h</code>	<code>(false)</code>
<code>--values</code> Shows the values of all command line options and parameters.	<code>-V</code>	<code>(false)</code>
<code>--stat-server</code> Starts the statistic server, which is a simple TCP/IP server showing information about the decomposition process and allows a basic control over running DESIJ processes.	<code>-E</code>	<code>(true)</code>
<code>--silent</code> Suppresses the starting message.	<code>-Z</code>	<code>(false)</code>

<code>verbose=[0 1 2 3]</code>		(1)
Writes detailed information of what is happening at the moment to the command line. 0: no information, 3: nearly every operation.		

<code>--punf-mpsats-output</code>	<code>-m</code>	(false)
Shows PUNF and MPSAT output for debugging.		

<code>--gui</code>	<code>-G</code>	(false)
Starts the graphical user interface.		

<code>--commandline</code>	<code>-C</code>	(false)
Opens an interactive command line for STG editing.		

<code>--productive</code>	<code>-v</code>	(true)
Several optimisations take place. Essentially, the places are just numbered after contractions. This makes the result more readable and accelerates the computations.		

<code>operation=[bisim check cl convert create decompose info info1 info2 killdummies rg redde1 show]</code>		(decompose)
Determines the working mode for DESIJ as follows:		
<code>bisim &lt;specification&gt; &lt;components&gt;</code>		
Finds and prints an STG bisimulation between the STG in <specification> and the parallel composition of the STGs in the <components> files. If an STG contains dummies, they are automatically contracted.		
<code>check &lt;specification&gt; &lt;components&gt;</code>		
Same as for <code>bisim</code> , but only checks if a STG-bisimulation exists. It is also faster than <code>bisim</code> .		
<code>cl &lt;file1&gt;</code>		
Opens the interactive command line mode.		
<code>convert &lt;file1&gt; &lt;file2&gt; ...</code>		
Converts the input STG to a different format. See <code>format</code> parameter for more details.		
<code>create</code>		
Creates a predefined STG model. When using this, the <code>model</code> parameter is mandatory.		
<code>decompose &lt;file1&gt; &lt;file2&gt; ...</code>		
Decomposes the given STGs. This operation is affected by various other parameters and options.		
<code>info,info1,info2 &lt;file1&gt; &lt;file2&gt; ...</code>		
Prints several infos of the given STGs. <code>info2</code> gives the most detailed ones.		
<code>killdummies &lt;file1&gt; &lt;file2&gt; ...</code>		
Contracts all dummy transitions, if possible.		
<code>rg &lt;file1&gt; &lt;file2&gt; ...</code>		
Creates the reachability graph.		
<code>redde1 &lt;file1&gt; &lt;file2&gt; ...</code>		
Deletes all redundant places.		
<code>show &lt;file1&gt; &lt;file2&gt; ...</code>		
Converts the STG to PS and opens a viewer.		

## 6.2 Control of the Decomposition Algorithm

<code>version=[basic csc-aware lazy-multi lazy-single tree]</code>		(tree)
The decomposition strategy to use.		

<code>--aggregation</code>	<code>-a</code>	(false)
Performs tree aggregation. Works only for tree decomposition and CSC-aware decomposition.		

<code>--leave-dummies</code>	<code>-l</code>	(false)
Do not backtrack for dummy transitions which are not contractable due to structural reasons.		

<code>max-csc-backtracking=[1..MAX.INT]</code>		
When CSC aware decomposition is enabled, this value determines the maximum number of nodes for which it is tried to solve CSC with signal of the STG before CSC is solved with new signals.		
<code>mcs=[1..MAX.INT]</code>		(10)
The maximum number of signals a component can have to perform component aggregation.		
<code>--recontract</code>	<code>-#</code>	(true)
After solving CSC with known signals, the signals not destroying CSC cores are contracted again.		
<code>--stop-when-backtracking</code>	<code>-A</code>	(false)
Stops decomposition if the first structural auto-conflict occurs. Mainly for debugging.		
<code>--risky</code>	<code>-Y</code>	(false)
When true, structural auto conflicts are ignored.		
<code>conflict-signal-exception</code>		
Which signals are treated different in case of an autoconflict (see <code>conflict-strategy</code> ). Turns conservative to risky and vice versa.		
<code>partition</code>		(finest)
Partition of output signals for components. For instance, <code>partition=x:y.z1:z2.ack</code> will create the three components (x,y), (z1,z2) and (ack). <code>partition=finest</code> automatically determines the finest possible partition.		
<code>--incomplete-partition</code>	<code>-i</code>	(false)
Checks the given partition only for conflicts between outputs and not if it is complete. i.e. all outputs occur within.		
<code>--remove-redundant-transitions</code>	<code>-T</code>	(true)
Redundant transitions are removed during decomposition.		
<code>--order-dummy-transitions</code>	<code>-o</code>	(true)
Dummy transition contractions are ordered ascending by the number of newly generated places.		
<code>--postpone-contractions</code>	<code>-p</code>	(true)
If a transition contraction is not possible it is tried later again, as long as there are other contractions possible.		
<code>--forbid-self-triggering</code>	<code>-s</code>	(true)
Consider transitions as non-contractable if their contraction leads to a self-triggering situation and the respective places are not redundant.		
<code>--safe-contractions</code>	<code>-f</code>	(true)
Performs only safeness preserving contractions.		
<code>--safe-contractions-unfolding</code>	<code>-x</code>	(false)
Check safeness preserving contractions with MPSAT.		
<code>deco-tree=[combined random top-down]</code>		(combined)
The method used to generate the decomposition tree.		

<code>max-unfolding-size=[0..MAX_INT]</code>	(100)
The maximum size of an STG ( <code>#Transition + #Places</code> ) for which properties are checked on the unfolding.	

### 6.3 Handling of Redundant Places

When the deletion of redundant places is activated, it is checked for (in this order):

- easy special cases (e.g. a place with empty preset is always redundant)
- loop only places
- duplicate places
- shortcut places
- implicit places (with unfoldings)

The latter two possibilities are only checked if they are enabled (see below), the last one only if the STG is small enough, see also parameter `max-unfolding-size`.

<code>--remove-redundant-places</code>	<code>-P</code>	(true)
Redundant places are removed during decomposition.		

<code>--shortcutplace</code>	<code>-u</code>	(true)
Check also for shortcut places when looking for redundant places.		

<code>shortcut-length=[2..MAX_INT]</code>	(MAX_INT)
Maximal path length for shortcut places.	

<code>--red-unfolding</code>	<code>-X</code>	(false)
Use MPSAT to check for implicit places. When this option is enabled, every implicit place is found.		

<code>--check-red-often</code>	<code>-O</code>	(false)
It is checked for redundant places (and transitions) before every contraction. Reduces performance significantly.		

<code>max-place-increase=[1.0..MAX_FLOAT]</code>	(1.1)
If during the contraction of some transitions the number of places exceeds the original number multiplied by this value, redundant places are deleted before it is proceeded with the contraction. Prevents the ‘explosion of places’ during contraction.	

### 6.4 Synthesis of Components

<code>--synthesis</code>	<code>-y</code>	(false)
When enabled DesiJ tries to synthesise the components with an external tool.		

<code>equations</code>	(equations)
The file in which the equations are stored if synthesis is enabled.	

<code>syn-tool=[mpsats]</code>	(mpsats)
The given tool is used for synthesis of the components.	

`syn-param=[arbitrary]`

The following parameters are forwarded to the synthesis tool.

## 6.5 Various Options

`hide`

The set of hidden signals when checking for bisimilarity.

`model=[art|seq|par|multipar|seqpartree|parseqtree]`

When the operation is `create`, this parameter defines the generated STG model.

`--hide-internal-handshakes`

`-H` `(true)`

When creating models out of handshake components, the internal handshakes are hidden, i.e. labelled with `lambda` in this case.

`--handshake-component-csc`

`-c` `(false)`

When creating models out of handshake components, the components are generated with CSC.

`predef=[advanced|original]`

Predefined parameter/option values.

## 6.6 Output Options

`--write-logfile`

`-L` `(true)`

Writes a logfile.

`logfile`

`(desij.logfile)`

Name of the logfile.

`--intermediate-results`

`-I` `(false)`

Intermediate results are written to disk. Mainly for debugging.

`format=[g|dot|ps]`

`(g)`

Fileformat of the written STGs. `*.g` is the standard exchange format for STGs, `*.dot` is a dot (from GraphViz) source file for the generation of graphical representations, `*.ps` is a PostScript file.

`--save-all-places`

`-S` `(false)`

When saving in `*.g` format, all places are saved explicitly.

`--redel-before-save`

`-b` `(false)`

Removes redundant places and transitions before an STG is saved. Not needed for decomposition.

`--reachability-graph`

`-R` `(false)`

Writes the reachability graph of an STG instead of the STG itself.

`label`

User defined label/caption for output in graphic formats.

`outfile`

Name of output file for operations different from `decompose`.

## References

- [KK01] V. Khomenko and M. Koutny. Towards an efficient algorithm for unfolding Petri nets. In K.G. Larsen and M. Nielsen, editors, *CONCUR 2001*, LNCS 2154, 2001.
- [SVWK06] M. Schaefer, W. Vogler, R. Wollowski, and V. Khomenko. Strategies for optimised STG decomposition. In *Proc. ACSD'06*, 2006.
- [VK05] W. Vogler and B. Kangsah. Improved decomposition of signal transition graphs. In *Proc. ACSD 2005*, pages 244–253, 2005.
- [VW02] W. Vogler and R. Wollowski. Decomposition in asynchronous circuit design. In *Concurrency and Hardware Design*, LNCS 2549, pages 152 – 190. Springer-Verlag, 2002.