The Package ω

Solving of Parity Games over Finite Graphs
and Application to μ-calculus
model-checking

Dominik Schmitz and Jens Vöge

RWTH-Aachen
Lehrstuhl für Informatik VII
Prof. Dr. Wolfgang Thomas

Contents

Chapter 1

Introduction

This toolkit provides an implementation and a user interface to an algorithm to solve parity games. The algorithm is described in [VJ00] and the implementation in [SV00]. Furthermore this toolkit contains a front-end to use the connection between solving parity games and model-checking for the modal μ-calculus. For this part the following reference might be helpful [EJS93].

In the following a very short overview of the problem of *solving parity games* is given. Therefore after a motivation at first parity games are introduced and explained with a small example. For a description of the algorithm, we have to refer you to the paper mentioned above. Furthermore it is shown at the example, how the toolkit is used and this is done also for a small model-checking problem.

1.1 Motivation

The package OMEGA of algorithms in ω-automata theory should support the researcher in getting experience with the performance of such algorithms and in carrying out case studies. Actually this package addresses only one problem from ω-automata theory which belongs to the theory of infinite games: the construction of winning strategies in parity games. These games have a direct connection to the model-checking problem for the modal μ-calculus (cf. [EJS93]). One of the central open questions in the field asks whether this model-checking problem or, equivalently, the construction of winning strategies in parity games is possible in polynomial time.

In [VJ00] a new algorithm was developed, which has the potential of being more efficient than previously known procedures (e.g., [EJ91, McN93, Jur00]). The new algorithm follows the idea of strategy improvement as devised by Puri [Pur95], but now in a completely discrete setting rather than over a dense value domain. The algorithm is hard to analyze; indeed, it is open whether the running time of this concrete algorithm can be bounded by a polynomial. Furthermore, the execution of the algorithm is hard to follow since it induces in each step a global change over the given game graph. In this situation, it is helpful to study the algorithm with an implementation and some auxiliary functions which help in assessing the main characteristics of its behaviour.

1.2 Parity games

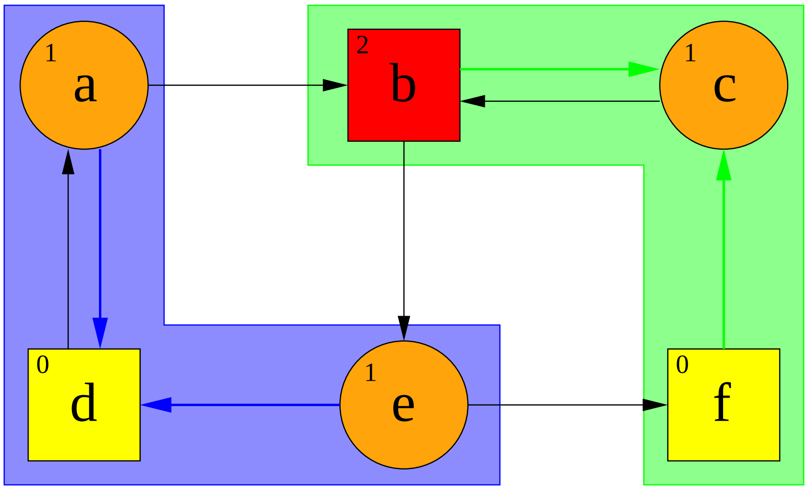
A parity game is an infinite game played on a finite colored graph (the "arena" of the game). The game graph is of the form $G = (V_0, V_1, E, c)$. The set V of vertices is the union of the two disjoint sets vertex V_0 and V_1, and we have $E \subseteq (V_0 \times V_1) \cup (V_1 \times V_0)$, i.e., an edge leads from a V_0-vertex to a V_1-vertex or vice versa. We assume $vE \neq \emptyset$ for all vertices v.

A play from vertex v is an infinite sequence $v_0 v_1 v_2 \ldots$ of vertices with $(v_i, v_{i+1}) \in E$ for all $i \geq 0$; one imagines players 0 and 1 moving a token along edges through the graph in a nonterminating sequence of steps. If $v_i \in V_0$ it is the turn of player 0 to decide for an edge leaving v_i, otherwise player 1 has to choose an edge leaving v_i.

The winner of a play $v_0 v_1 v_2 \ldots$ is defined with a reference to the component c of the game graph, which is a function (called coloring) $c : V \to \{0, \ldots, k\}$ for some k. The play $v_0 v_1 v_2 \ldots$ is won by player 0 if in the associated sequence $c(v_0)c(v_1)c(v_2)\ldots$ of colors the minimal color occurring infinitely often is even; otherwise player 1 wins the play.

It is a well-known theorem of the theory of infinite games (see e.g. [EJ91, Tho97]) that parity games (even over infinite game graphs) are "determined" in the following strong sense: From each vertex either player 0 or player 1 can force a win by a *positional winning strategy*. More precisely, the vertex set is partitioned in two *winning regions* W_0, W_1 such that player 0 wins starting from any vertex in W_0 by a fixed choice of outgoing edges for the V_0-vertices in W_0, and analogously player 1 wins starting from any vertex in W_1 by a fixed choice of outgoing edges for the V_1-vertices in W_1. The "solution of the game" is given by the sets W_0, W_1 and corresponding edge sets $E_0 \subseteq (V_0 \cap W_0) \times V_1$ and $E_1 \subseteq (V_1 \cap W_1) \times V_0$. As shown by [EJS93], the problem of solving parity games over finite graphs is in NP $\cap$ co-NP and polynomially equivalent to the model-checking problem for the full modal μ-calculus. All known algorithms [EJ91, McN93, Jur00] have an exponential worst case running time. A key problem of the theory of program verification is to settle the question whether parity games can be solved in polynomial time.

Small example The following small game graph with only six vertices and three colors might help to make you understand parity games.



The winning strategies and winning regions are already marked. From the vertices within the green box player 1 wins with the strategy given, because every play starting in one of these three vertices ends in the loop consisting of the vertices `b` and `c` with the smallest color being `1` and therefore winning for player 1. Every play starting from the vertices in the blue box ends in the loop `a, d` with the smallest color `0` and so player 0 wins with the given strategy.

1.3 Working with the toolkit

To find the solution to the small example above with the toolkit the user has to do the following:

First at all the software has to be installed the right way as it is described in the appendix, see appendix A. Then the Lisp interpreter must be started and the file `omega-0.1.1.lsp` must have been loaded. If this is all done, then the user can type in the following at the Lisp interpreter prompt:

```
> (c-parity-1arg
     (parity-game (vertex ('a) 0 1) (vertex ('b) 1 2)
                  (vertex ('c) 0 1) (vertex ('d) 1 0)
                  (vertex ('e) 0 1) (vertex ('f) 1 0)
                  (edge ('a) ('b)) (dedge ('a) ('d))
                  (edge ('b) ('e)) (dedge ('b) ('c))
                  (edge ('e) ('d)) (edge ('e) ('f))
                  (edge ('f) ('c))))
```

After a short time the following output appears in the Lisp interpreter shell:

```
((D) (A) (E)) ;
((F) (C) (B)) ;
(((B) (C)) ((E) (D)) ((C) (B)) ((A) (D)) ((F) (C)) ((D) (A)));
2 ;
(0.01 0.0)
>
```

The first list (up to the semicolon) gives the winning region of player 0, the second the winning region of player 1, and the third the strategies for both players. If there is an entry for a player from a vertex, which does not belong to the winning region of the player this edge can be understood as "the best answer" although leading to loose the game.

The next item of the results gives the number of iterations the algorithm needed and the following last item gives the real time (user and system) the algorithm needed.

1.4 Model-checking with the modal μ-calculus

```
(nu y (mu x (box T (or (and (diamond "send" (and)) y) x))))
```

This is an example for a modal μ-calculus formular, which can be tested on a named transition system, a so called Kripke structure. The question is whether the formular holds in a specified state of the structure or not, therefore the functions of the package SIA-MU-CALCULUS simply return the list of states, in which the complete formular holds and otherwise `NIL`. Most times the transition systems are very large therefore we do not want to give a picture of one here, but simply say we use the Two-Link-Network (size about 1600 vertices) and assume that this transition system is safed in the variable `graph`. The formular is assumed to be in the variable `phi6`. Then we get the following results, if we let the algorithm run on this input:

```
> (eval-form3 graph phi6)
..
NIL
```

and for more detailed information on the run of the algorithm:

```
> (run-form3 graph phi6)
..

Real time:  65.41632 sec.
Run time:  55.39 sec.
Space:  43665712 Bytes
GC: 64, GC time:  2.67 sec.
NIL ;
2 ;
(8.51 0.01)
```

As already explained the result `NIL` says that the complete forumlar is not valid in any state of the Two-Link-Network. The last two items are of course number of iterations and real-time again. The information before gives not the running time only of the algorithm, but the whole running time including the conversion from the model-checking problem to the parity game.

Chapter 2

Documentation

The documentation contains two parts:

- One which lists and explains the data types which occur in the implementation and
- one which lists the packages and explains all the exported functions of a package.

The whole system $\boldsymbol{\omega}$ consists of seven Common Lisp packages: OMEGA, CCS, SIA-PARITY-GRAPH, SIA-STRATEGY, SIA-GENERATOR, SIA-BASIS and SIA-MU-CALCULUS.

The package OMEGA contains a loader that loads the newest versions of all other packages. The SIA-BASIS package contains some basic functions and macros that are used by the other packages.

The core package of the system is SIA-STRATEGY. It is a wrapper around a C program that does strategy synthesis. The package SIA-PARITY-GRAPH provides an environment to describe parity graphes in a simple way. It converts this representation into the internal one. The package SIA-GENERATOR contains functions that generate some special parity games of scalable size. A front-end for model-checking problems is provided by package SIA-MU-CALCULUS. It also allows to use CTL operators. By the main function in this package a model-checking problem is transformed into a parity game. Finally the package CCS contains a front-end for CCS terms. It can transform a given set of CCS-terms into a transition system that can be used for the model-checker front-end.

2.1 Data types

This section describes the data types which occur in this implementation. There are several different data types for the representation of graphs, but also types and structures to describe for example μ-calculus formulars or ccs equations.

If there is nothing said about the structure, a list, maybe nested is most times supposed to be the underlying data structure, e.g. for μ-calculus formulars and ccs equations. Although they are not really new data types, descriptions for these types can be found here as well.

Name

In **CCS-Terms and -Actions** the following keywords can be used to define a system of CCS equations: `tau`, `bar`, `act`, `par`, `sum`, `restrict`, `subst`.

Synopsis

CCS-Action

`tau`

lexp

`(bar` *lexp*`)` where *lexp* is a Lisp Term that is not `tau` and not a list with `bar` as its first element.

CCS-Term

`P`

`(act` *action CCS-Term*`)`

`(sum` *CCS-Term CCS-Term*`)`

`(par` *CCS-Term CCS-Term*`)`

`(restrict (`*action* `...)` *CCS-Term*`)`

`(subst ((`*old-action new-action*`) ...)` *CCS-Term*`)`

Description

A system of CCS equations can be used to describe an edge labeled graph. These keywords are allowed to define such a system of CCS equations.

Parameter

Return value

Environment

Package

These keywords belong to the package CCS.

Implementation

Bugs / Errors

See also

Name

Within μ-**calculus formulars** the following keywords can additionally be used: `mu`, `nu`, `box`, and `diamond`.

Synopsis

(mu *variable formular*)
(nu *variable formular*)
(box *label formular*)
(diamond *label formular*)

Description

These names are only exported as keywords, there are no functions or macros named like this.

Furthermore in μ-calculus formulars the use of the keywords `and` and `or` is allowed. Because the modal μ-calculus can only cope with monotonic formulars the use of `not` is restricted. See the related functions for details.

Parameter

formular A subformular in modal μ-calculus terms.

variable An atomic name for a variable used in *formular*.

label This has to be a label of the Kripke structure or NIL for all labels.

Return value

Environment

Package

These keywords belong to the package SIA-MU-CALCULUS.

Implementation

Bugs / Errors

See also

eval-form2, eval-form3, run-form3

Name

`graph pairs` represents the transition relation of a graph by a list of pairs, where each pair is an edge of the graph.

Synopsis

`((source_1 target_1) (source_2 target_2)...(source_n target_n))`

Description

This data type is the input type for the central access function `c-parity` to the C-program. Therefore every function calling this function must be able to handle this data type.

Labeled edges are not supposed to be handled with this data structure or are at least not handled with in this implementation.

Parameter

`source_1,source_2,...target_n` have to be valid names of vertices i.e. tuples.

Environment

Package

This data structure occurs in various packages (the reason is given above): SIA-BASIS, SIA-GENERATOR, SIA-MU-CALCULUS, SIA-PARITY-GRAPH, SIA-STRATEGY.

Implementation

Bugs / Errors

See also

`graph alist, graph hash table, graph adjacents vector`

Name

`graph alist` is used to represent the transition relation of a graph by an association list.

Synopsis

```
((source_1 (target_1_1 ... target_1_n))
 ...
 (source_p (target_p_1 ... target_p_m)))
```

or

```
((source_1 ((label_1 (target_1_1_1 ... target_1_1_n))
            (label_2 (target_1_2_1 ... target_1_2_n'))
            ... ))
 ...
 (source_p ((label_1 (target_p_1_1 ... target_p_1_m))
            (label_2 (target_p_2_1 ... target_p_2_m'))
            ... )))
```

Description

This data type is also used to represent the transition relation but this time an entry of the list consists of a list, where the first element is the source of all edges to the vertices in the second element.

There is also the possibility that the transitions are named i.e. the second element is again a list of lists, where the first element is the name of the transition and after that the adjacent vertices follow.

For the transform functions `pair-to-assoc` and `assoc-to-pair` the association lists must not have labels.

Parameter

`source_*, target_*` have to be valid names of vertices i.e. tuples.

`label_*` has to be a valid Lisp term.

Environment

Package

This data structure occurs in the packages: CCS, SIA-BASIS, SIA-MU-CALCULUS.

Implementation

Bugs / Errors

See also

`graph pairs, graph hash table, graph adjacents vector`

Name

`graph hash table` is used to represent the transition relation of a graph by a hash table.

Synopsis

A hash table is created by using the command `make-hash-table`. Most time it is necessary to set the test-function to `#'equal`, because the default `#'eq` is not sufficient.

An entry in the hash table can be searched and modified by using `(gethash` *`key hash-table`*`)`. To modify it, `setf` has to be used.

Description

This data type is similar to the association list but much more efficient for searching. An entry of a hash table consists of a key (the first element of an entry of an association list) and a value (the second element).

Of course the value can contain named transitions or not just the same as with the association list.

Parameter

See `graph alist` for details.

Environment

Package

This data structure occurs in various packages: CCS, SIA-BASIS, SIA-MU-CALCULUS, SIA-STRATEGY.

Implementation

Bugs / Errors

See also

`graph alist, graph pairs, graph adjacents vector`

Name

`graph adjacents vector` is used to represent the transition relation of a graph by an adjacents vector.

Synopsis

```
#'((target_1_1 ... target_1_n)
   ...
   (target_p_1 ... target_p_m))
```

Description

This data type is again similar to the association list and therefore simiiliar to the hash table. It is also efficient under the assumption that the vertices are named by numbers, because the index of the vector is handled as the source. Of course therefore the entries should be numbers as well.

Although it is possible to cope with labels here too, it does not appear in this implementation (hope I'm right).

I.e. the index of the vector is used as the key (the first element of an entry of an association list) and the entry at this position as the value (the second element).

Parameter

See `graph alist` for details.

Environment

Package

This data structure occurs in the packages SIA-BASIS, SIA-STRATEGY, and SIA-MU-CALCULUS(truth-converter).

Implementation

Bugs / Errors

See also

`pairs-to-adjacents-vector, graph alist, graph pairs, graph hash table`

Name

`alphabet` is used to have an easy assignment between symbols and numbers.

Synopsis

(make-alphabet '(*sym_1* ... *sym_n*))

(symbol-no *alphabet symbol*)

(nth-symbol *alphabet n*)

Description

This data type represents a set of unique symbols and gives a unique number to each symbol. Therefore the numbers can be used instead of the symbols. The corresponding functions deliver either the symbol to a number or the number to a given symbol.

This data type is used to shorten the names of vertices in a parity game.

Parameter

`sym_1, ... , sym_n` The parameters to the creator of this structure `make-alphabet` is a list of unique symbols.

`alphabet` An alphabet structure as created with `make-alphabet`.

`symbol` A symbol of the alphabet.

`n` An integer number in the appropriate range.

Environment

Package

This data structure occurs in the packages SIA-BASIS, SIA-STRATEGY, and SIA-MU-CALCULUS.

Implementation

Bugs / Errors

See also

`make-alphabet, symbol-no, nth-symbol, alphabet-size`

2.2 Functions

The functions and macros are divided into several packages, seven up to now:

OMEGA	is the central package.
CCS	contains functions to create named transition systems.
SIA-BASIS	provides basic functions, which can also be used in other Lisp programs.
SIA-GENERATOR	contains several generators for scalable parity-games.
SIA-MU-CALCULUS	provides an interface to work with modal μ-calculus formulars.
SIA-PARITY-GRAPH	contains macros and functions to define parity games easily.
SIA-STRATEGY	provides the connection to the C-program.

The descriptions of the functions are divided into the packages as well to have all the functions concering the same topic closer together. For an alphabetic order please visit the index register.

The central package is the package OMEGA, which loads all the other packages, when activated. Therefore this package should be loaded to work with this framework to examinate parity games. For a more detailed description of a package you can read the introduction to it in the following sections.

2.2.1 Package OMEGA

As already mentioned this package is the central one, because it loads the other ones, by searching the newest versions of the corresponding files. I.e. in this file the connection between the logical package name and the somehow physical file name is determined.

The package exports only the function `latest-version`. When loaded this function is used to find the latest versions of the given packages and load them. It expects the files to have the extension ".lsp" and the predefined beginning. The numbering should be `x.y.z` where `x`, `y`, and `z` are positive integers or zero.

Name

`latest-version` finds the file with the highest version number starting with *name* and ending with the given extension.

Synopsis

`(latest-version` *name extension*`)`

Description

As already explained the latest file with the appropriate beginning and extension is found and its name is returned.

Parameter

name The beginning of the file the latest-version is searched for.

extension The extension of the file.

Return value

The complete name of the latest version including version number.

Environment

Package

This function belongs to the package OMEGA.

Implementation

Bugs / Errors

See also

2.2.2 Package CCS

This package contains functions to cope with CCS-Terms.

To use the function `ccs-gen` to generate a parity game from a system of CCS-equations, one should have such a system. Examples can be found in the file 'ABP.lsp'.

The graph should be simplified, before the C-program is invoked with it. The function `simplify-graph` from the package SIA-BASIS can be used for this purpose.

Name

`ccs-gen` generates an edge labeled graph from a system of CCS equations.

Synopsis

`(ccs-gen` *`initial-process equations`*`)`

Description

A system of CCS equations can be used to describe an edge labeled graph. Starting from the *`initial-process`* all global states and transitions of the system are generated.

The function does not check whether the second parameter is a wellformed CCS equation system.

Parameter

`initial-process` Identifier of initial process.

`equations` List of CCS-equations.

Return value

The return value is a hash table. For each vertex there is an entry in the hash table. The key is the name of the vertex and the value is an association list describing the edges leaving this vertex. Each key of the association list is an edge label and the corresponding value the list of vertices reachable by edges with this label.

Environment

Package

This function belongs to the package CCS.

Implementation

Bugs / Errors

See also

2.2.3 Package SIA-PARITY-GRAPH

This package includes several macros to simplify the definition of parity games in Lisp terms.

There are some constructs with PASCAL-like syntax to support users not familiar with Lisp. The three most important macros to define vertices and edges of a parity graph: `vertex`, `edge`, and `dedge`. These macros are locally defined to the macro `parity-game` which collects the definition items to compose the complete game and to provide the arguments for the call of a function which computes the strategy.

Name

`parity-game` constructs a parity-game from a high-level description.

Synopsis

(parity-game *body*) (*macro*)

Description

In the high-level description of the parity-game the macros `edge`, `dedge`, `vertex`, `begin`, or `for` should be used. `parity-game` returns the constructed game-graph in the form, which is needed for the function computing the strategy `c-parity-1arg`.

Parameter

body A high-level description of the parity-game-graph using the macros `edge`, `dedge`, `vertex`, `begin`, and `for`

Return value

Multiple values are returned by the macro: First a vector containing the vertices of player 0, secondly a vector containing the vertices of player 1, thirdly a list of pairs representing the edges in the parity-game graph and finally the parity-list, where a list-entry enumerates the vertices belonging to a color. The vertices listed in the first list-entry have the smallest color 0.

Environment

Package

This macro belongs to the package SIA-PARITY-GRAPH.

Implementation

Bugs / Errors

See also

`edge`, `dedge`, `vertex`, `begin`, `for`, `c-parity-1arg`

Name

`begin` is used as a synonym for `progn`.

Synopsis

`(begin` *`body`*`)` (*macro*)

Description

A more common (?) synonym for `progn` in PASCAL-style which always produces the return value `nil`. Mostly used in order to define a parity-game with the macros `vertex`, `edge` and `dedge`.

Parameter

`body` A high-level description of the parity-game using the macros `edge`, `dedge`, `vertex`, `begin`, and `for`

Return value

`nil`

Environment

Package

This macro belongs to the package SIA-STRATEGY.

Implementation

Bugs / Errors

See also

`for`

Name

`for` works like a 'normal' i.e. PASCAL-style for-loop.

Synopsis

`(for (`*`var from to`* `[`*`step`*`])` *`body`*`)` (*macro*)

Description

The reason for this macro being included in this package is to have a PASCAL-like and therefore maybe more common syntax for the loop construct `for`. The synopsis should be self-explaining.

Parameter

var The name of the index variable used in the loop.

from The first value of the index variable.

to The limit value which causes termination of the loop, when reached or exceeded.

step This optional argument should be a number diverse zero, by which the value of the index variable is changed each iteration.

body The statement(s) to be iterated contain `vertex`, `edge`, or `dedge` for example.

Return value

The return value of the body is handed over.

Environment

Package

This macro belongs to the package SIA-PARITY-GRAPH.

Implementation

Bugs / Errors

See also

`begin`, `vertex`, `edge`, `dedge`

Name

`vertex` declares a new vertex of a parity-game.

Synopsis

(vertex *tuple player index*) (*macro*)

Description

For the declaration of a new vertex of a parity-game there are three information items to mention: The name of the vertex (a tuple), the player it belongs to and the index in the parity-list.

The name of the vertex must be a list or at least bracketed.

Execution of the expanded function causes the vertex to be inserted in the vector of vertices belonging to the right player, and in the parity-list at the right place.

Parameter

tuple A list or bracketed atom, which gives a name to the vertex.

player The number of the player (i.e. 0 or 1) the vertex belongs to.

index The index numbering of the vertex in the parity-game.

Return value

The return value is not relevant, it is the index of the vertex, maybe.

Environment

Package

This macro belongs to the package SIA-PARITY-GRAPH.

Implementation

The definition of this macro is local to `parity-game` and can therefore be used only in the parameter `body` of this macro.

Bugs / Errors

See also

`begin`, `for`, `edge`, `dedge`

Name

`edge` declares a new edge of a parity-game.

Synopsis

`(edge` *`source target`*`)` (*macro*)

Description

For the declaration of a new edge of a parity-game the names of the two vertices to be connected are needed. The requirements are the same as in the declaration of a new vertex: The name of a vertex must be a list or at least bracketed.

Execution of the expanded function causes the edge to be inserted into the list of transitions in the game-graph representation.

Parameter

`source` A list or bracketed atom, which gives the name of a vertex.

`target` The same for the target vertex.

Return value

The return value is not relevant, it is the list of transitions inserted so far.

Environment

Package

This macro belongs to the package SIA-PARITY-GRAPH.

Implementation

The definition of this macro is local to `parity-game` and can therefore be used only in the parameter `body` of this macro.

Bugs / Errors

See also

`begin`, `vertex`, `for`, `dedge`

Name

`dedge` declares two new edges of a parity-game.

Synopsis

`(dedge` *`source target`* `)` (*macro*)

Description

This command creates two new edges connecting the given vertices in both directions. The requirements are the same as in the declaration of a new vertex: The name of a vertex must be a list or at least bracketed.

Execution of the expanded function causes the edges to be inserted into the list of transitions in the game-graph representation.

Parameter

source	A list or bracketed atom, which gives the name of a vertex.
target	The same for the target vertex.

Return value

The return value is not relevant, it is target, maybe.

Environment

Package

This macro belongs to the package SIA-PARITY-GRAPH.

Implementation

The definition of this macro is local to `parity-game` and can therefore be used only in the parameter `body` of this macro.

Bugs / Errors

See also

`begin`, `vertex`, `for`, `edge`

Name

`parity-game-vertex` corresponds to the functional body of the macro `vertex`.

Synopsis

(parity-game-vertex *vertices0 vertices1 parity-list tuple parity index*)

Description

See `vertex`.

Parameter

vertices0 The list of vertices belonging to player 0 so far.

vertices1 The same for player 1.

parity-list The classification of the vertices by index number so far.

tuple A list or bracketed atom, which gives a name to the vertex.

parity The number of the player (i.e. 0 or 1) the vertex belongs to.

index The index numbering of the vertex in the parity-game.

Return value

Seems to be the updated parity-list.

Environment

Package

This function belongs to the package SIA-PARITY-GRAPH.

Implementation

This function is no longer exported.

Bugs / Errors

See also

`vertex`

Name

`parity-game-dedge` corresponds to the functional body of the macro `dedge`.

Synopsis

(parity-game-dedge *edges source target*)

Description

See `dedge`.

Parameter

edges The list of transitions inserted so far.

source A list or bracketed atom, which gives the name of a vertex.

target The same for the target.

Return value

Seems to be the list of transitions.

Environment

Package

This function belongs to the package SIA-PARITY-GRAPH.

Implementation

This function is no longer exported.

Bugs / Errors

See also

`dedge`

2.2.4 Package SIA-STRATEGY

This package is the 'heart' of the user interface because it provides the interface to the C-program. The main functions to invoke the C-program with a parity game defined in Lisp terms and to interpret the results are gathered together here.

There are three functions to run the C-program which differ from their fields of application:

> The function `strategy` provides just a better name for `c-parity`, which is the central function. Finally the function `c-parity-1arg` simplifies the use of the generators and of the macros to define a parity game by changing/setting some default values.

The algorithm uses an inital strategy, which is improved step by step. Of course one can think of several, maybe 'good' initial strategies instead of the arbitrary one, which is generated while constructing the game. One of these maybe 'good' initial strategies is implemented in form of the function `initial-strategy-highest-reward`, which is used by default. The user might define new functions implementing an initial strategy, but these functions must have the same synopsis (except the name of course) as the one mentioned above.

Name

`c-parity` is the main function to run the algorithm written in C.

Synopsis

(c-parity *vertices-player-0 vertices-player-1*
transition-relation parity-list [*relevance-order*
[:initial-strategy *initial-strategy*]])

Description

`c-parity` causes the conversion of the game-graph into the form the C-program needs, runs the C-program and returns the result of it with the names used in the definition of the game graph in Lisp terms. There are also some files created, which contain more detailed information about the actual run of the program. Normally `strategy` or `c-parity-1arg` are used to invoke this function, but the functions concerning the modal μ-calculus call this function directly.

Parameter

vertices-of-player-0 The sequence of vertices belonging to player 0.

vertices-of-player-1 The same for player 1.

transition-relation The list of edges of the game. An edge is a list of two vertices.

parity-list The coloring of the game graph, consisting of a list of lists of vertices.

relevance-order This optional argument gives the order of the parity-list: `:first-best` says the first entry of the list is best for player 0 (this is default for c-parity1-arg and cannot be changed by an argument to this macro), `:last-best` is the default value and says analogously the last entry in the parity-list is best for player 0.

initial-strategy To use this optional argument one has to write `:initial-strategy` in front of it, because it is a key argument. The parameter has to be the name of a function, which is defined similiar to `initial-strategy-highest-reward`, which is the default function. Use `NIL` instead to inhibit any initial-strategy, i.e. the arbitrary strategy inherently created while building the game is used.

Return value

Various pieces of information are returned in form of multiple values:

win0 The list of vertices from which player 0 wins the game.

win1 The same for player 1.

strategy A list, giving the best strategy for the players to play. For vertices belonging to the winning set of the player this is the winning strategy, for the others the best answer (although resulting in losing the play).

iterations The number of iterations, which gives the number of executions of the main loop of the C-program.

time-needed The result of the shell-function time applied to the call of the C-program, which returns the time the program needs overall. It is represented as a pair of user time and time for system calls.

Environment

This function uses the variables `path-to-executing-dir` and `parity-synthesis-executable` to find the directory and to know the name of the C-program.

Package

This function belongs to the package SIA-STRATEGY.

Implementation

The exchange of information between the Lisp frontend and the algorithm written in C is managed by temporary files, which use the id of the clisp-process to be unique.

Bugs / Errors

See also

`strategy`, `c-parity-1arg`, `initial-strategy-highest-reward`

Name

`c-parity-1arg` is a simplified call of the function `c-parity`.

Synopsis

`(c-parity-1arg` *`generator`*
`[:initial-strategy` *`initial-strategy`*`])` (*macro*)

Description

As already mentioned this macro is thought of to be a simplified call of the function `c-parity`, which is suitable for games defined by a function, which returns the appropriate arguments for `c-parity`. The macro `parity-game` does this, but especially also all the functions in the package SIA-GENERATOR are such functions, so this macro simplifies the examination of the algorithm onto this (scalable) examples.

Parameter

`generator` The function call (including arguments) of a generator function for a parity game, returning multiple values: vertices-of-player0, vertices-of-player1, transition-relation, and parity-list. The parity-list is assumed to be ordered first-best.

`initial-strategy` This optional parameter must be a function determining the initial strategy of player 0 in the game, just like the function `initial-strategy-highest-reward`, which is default of this parameter. If the user wants to change the default, (s)he has to use the keyword `:initial-strategy` in front of the function parameter, which can also be `NIL`.

Return value

Of course the return value of `c-parity` is returned.

Environment

Package

This macro belongs to the package SIA-STRATEGY.

Implementation

The optional argument `relevance-order` of `c-parity` in the inner call is set to `:first-best`.

Bugs / Errors

See also

`strategy`, `c-parity`, `initial-strategy-highest-reward`

Name

`initial-strategy-highest-reward` determines the initial strategy, which is chosen.

Synopsis

`(initial-strategy-highest-reward` *`adjacents-vector winner-0`*`)`

Description

This function is only exported to give the user the possibility to say whether this inital strategy (or maybe another) should be used or not i.e. the arbitrary strategy originated in the construction of the parity-game is used.

Therefore the parameters should be ignored, but of course they are important, if the user wants to define a new function determining an initial strategy for player 0.

Parameter

`adjacents-vector` A vector with an entry for each vertex in the game. The entry consists of a list containing all the vertices the vertex is adjacent to.

`winner-0` A vector with an entry for each vertex in the game. There are only two entries allowed: `T` or `NIL`. The entry has to be `T`, if the vertex is good for player 0 otherwise it has to be `NIL`.

Return value

The return value is a new adjacents-vector, where the first element of each entry coincides with the new strategy.

Environment

Package

This function belongs to the package SIA-STRATEGY.

Implementation

Bugs / Errors

Maybe the two parameters do not deliver enough information for other strategies, so that the source code must be adapted to a more open definition of such functions.

See also

`c-parity-1arg`, `c-parity`

Name

`strategy` simply passes its arguments to the function `c-parity`.

Synopsis

`(strategy` *`parameters`*`)` (*macro*)

Description

The macro accepts nearly everything as its parameters, but only arguments, which fit to the syntax of `c-parity` make sense, because the macro only calls the function `c-parity` with the arguments committed.

Parameter

`parameters` Arguments, which fit to the definition of `c-parity`.

Return value

The return value is inherited from the return value of `c-parity`.

Environment

Package

This macro belongs to the package SIA-STRATEGY.

Implementation

Bugs / Errors

See also

`c-parity-1arg`, `c-parity`

2.2.5 Package SIA-GENERATOR

This package contains several generators for scalable examples of parity games, which are used to examine the asymptotic behaviour of the algorithm.

Mostly they are taken from earlier papers, where they were used to show that an proposed algorithm needs exponential time to find a solution. For none of these examples the present algorithm shows exponential behaviour. There are also often very regular forms in the improvement trace which is an output of the C-program that shows progress in finding a solution. Especially the number of iterations without stars is too low, because if the algorithm needs exponential time, then this has to be reflected by the number of iterations without stars.

For further information on the given generators you should have a look a the following descriptions and the corresponding papers if they exist.

Name

`gen-example` is a very small example for a generator template.

Synopsis

`(gen-example)`

Description

In this old version the macros `vertex` and `edge` are locally defined and the use is shown creating a small example graph, consisting of two vertices connected in both directions with each other.

Parameter

No parameters have to be given to the function.

Return value

The description of the game as demanded by the function computing the strategy.

Environment

Package

This function belongs to the package SIA-GENERATOR.

Implementation

Bugs / Errors

The local redefinition of the macros `vertex` and `edge` should be cancelled.

See also

`c-parity-1arg`

Name

`gen-example1` generates the bipartite version of the graph first described in a pre-version of [Jur00].

Synopsis

(gen-example1 *levels blocks*)

Description

This function describes a bipartite version of the graph, mentioned in a pre-version of the given paper. The definition uses only Lisp terms instead of the macros to define the game. There are various connections between levels and blocks, which are described in the pre-version of the paper.

Parameter

levels Factor to scale the graph, which gives the number of horizontal copies of a line (consisting of blocks).

blocks The number of vertical copies of a block, which is the smallest unit.

Return value

The adequate description of the game to be used as the input arguments of the function computing the strategy.

Environment

Package

This function belongs to the package SIA-GENERATOR.

Implementation

Again the hint that in this implementation the macros `parity-game`, `vertex`, `edge`, or `dedge` are **not** used.

Bugs / Errors

See also

gen-example1-b, gen-example1b, gen-example1c, gen-example10, c-parity-1arg

Name

`gen-example1-b` differs from `gen-example1` only in the way that the appropriate macros are used.

Synopsis

`(gen-example1` *`levels blocks`*`)`

Description

This function describes a bipartite version of the graph, mentioned in a pre-version of the given paper. The definition uses the macros to define the game. There are various connections between levels and blocks, which are described in the pre-version of the paper.

Parameter

`levels` Factor to scale the graph, which gives the number of horizontal copies of a line (consisting of blocks).

`blocks` The number of vertical copies of a block, which is the smallest unit.

Return value

The adequate description of the game to be used as the input arguments of the function computing the strategy.

Environment

Package

This function belongs to the package SIA-GENERATOR.

Implementation

This time the macros `parity-game`, `vertex`, `edge`, and `dedge` are used.

Bugs / Errors

See also

`gen-example1, gen-example1b, gen-example1c, gen-example10, c-parity-1arg`

Name

`gen-example1b` generates the (non-)bipartite version of the graph described in a pre-version of [Jur00].

Synopsis

`(gen-example1b` *`levels blocks`*`)`

Description

This function describes the non-bipartite version of the graph, mentioned in a pre-version of the given paper. The definition uses only Lisp terms instead of the macros to define the game. There are various connections between levels and blocks, which are described in the pre-version of the paper.

Parameter

`levels` Factor to scale the graph, which gives the number of horizontal copies of a line (consisting of blocks).

`blocks` The number of vertical copies of a block, which is the smallest unit.

Return value

The adequate description of the game to be used as the input arguments of the function computing the strategy.

Environment

Package

This function belongs to the package SIA-GENERATOR.

Implementation

Again the hint that in this implementation the macros `parity-game`, `vertex`, `edge`, or `dedge` are **not** used.

Bugs / Errors

See also

`gen-example1, gen-example1-b, gen-example1c, gen-example10, c-parity-1arg`

Name

`gen-example1c` generates the (non-)bipartite version of the graph first described in a pre-version of [Jur00], but using the appropriate macros.

Synopsis

`(gen-example1c` *`levels blocks`*`)`

Description

This function describes the non-bipartite version of the graph, mentioned in a pre-version of the given paper, but this time the appropriate macros `parity-game`, `vertex`, `edge`, and `dedge` are used. There are various connections between levels and blocks, which are described in the pre-version of the paper.

Parameter

`levels` Factor to scale the graph, which gives the number of horizontal copies of a line (consisting of blocks).

`blocks` The number of vertical copies of a block, which is the smallest unit.

Return value

The adequate description of the game to be used as the input arguments of the function computing the strategy.

Environment

Package

This function belongs to the package SIA-GENERATOR.

Implementation

This time the macros `parity-game`, `vertex`, `edge`, and `dedge` are used.

Bugs / Errors

See also

`gen-example1, gen-example1-b, gen-example1b, gen-example10, c-parity-1arg`

Name

`gen-example2` is another generator.

Synopsis

(gen-example2 *levels blocks chainlen*)

Description

A closer description of this generator will follow later.

Parameter

levels

blocks

chainlen

Return value

The adequate description of the game to be used as the input arguments of the function computing the strategy.

Environment

Package

This function belongs to the package SIA-GENERATOR.

Implementation

Some local macros `state` and `transition` are used to define the game.

Bugs / Errors

See also

`c-parity-1arg`

Name

`gen-example3` is another generator for a parity game.

Synopsis

(gen-example3 *levels blocks*)

Description

Further information will be available, when someone remembers this example.

Parameter

levels

blocks

Return value

The adequate description of the game to be used as the input arguments of the function computing the strategy.

Environment

Package

This function belongs to the package SIA-GENERATOR.

Implementation

The macros `vertex`, `edge`, and `dedge` are locally defined and used in the body of this function.

Bugs / Errors

See also

`c-parity-1arg`

Name

`gen-example4` is another generator for a parity game, maybe constructing the same game as `gen-example3`.

Synopsis

`(gen-example4` *`levels blocks`*`)`

Description

This generator seems to define the same game as `gen-example3`, but this time using the appropriate macros and infix-notation for arithmetic terms by defining abbreviations. A difference occurs within the third component of a vertex tupel. Everything else seems to be equal.

Parameter

`levels`

`blocks`

Return value

The adequate description of the game to be used as the input arguments of the function computing the strategy.

Environment

Package

This function belongs to the package SIA-GENERATOR.

Implementation

The appropriate macros (from the package SIA-PARITY-GRAPH) are used and abbreviations to simulate infix notation.

Bugs / Errors

See also

`gen-example3, c-parity-1arg`

Name

`gen-example5` generates a small and very simple parity game.

Synopsis

`(gen-example5)`

Description

This function creates a graph with 5 vertices named from 'a' to 'e' and some transitions. The result of computing the strategy is that player 0 wins from each vertex using the strategy ('b','a') and ('d','a').

Parameter

There are no parameters to explain.

Return value

The adequate description of the game to be used as the input arguments of the function computing the strategy.

Environment

Package

This function belongs to the package SIA-GENERATOR.

Implementation

The appropriate macros to define parity games are used.

Bugs / Errors

See also

`c-parity-1arg`

Name

`gen-example6` is able to generate the examples as described in [BLV96].

Synopsis

(gen-example6 *gamesize*)

Description

This generator can be customized not only by the parameter but also by inner functions (for example `c1,c2,d1,d2`), which determine the connections between the vertices. For further information we have to refer you to [BLV96].

Parameter

gamesize This parameter gives the number of vertices of the parity graph. Half of it belongs to player 0, the others to player 1. The parameter is tested to be divisible by 4 (depending on the inner functions used, sometimes it should even be divisible by 8 or 16).

Return value

The adequate description of the game to be used as the input arguments of the function computing the strategy.

Environment

Package

This function belongs to the package SIA-GENERATOR.

Implementation

There are inner functions `c1,..,c3,d1,..c3` used to determine connections between vertices. There are several possibilities implemented (but only one is uncommented). Of course new implementations of these functions can be added. The function itself checks for and ignores edges defined multiple times.

Bugs / Errors

See also

`gen-example6b, gen-example6c, c-parity-1arg`

Name

`gen-example6b` generates the same graph as `gen-example6` but using the appropriate macros.

Synopsis

`(gen-example6b` *`gamesize`*`)`

Description

This function differs from `gen-example6` only in the way the graph is defined. This time the appropriate macros are used.

Parameter

`gamesize` This parameter gives the number of vertices of the parity graph. Half of it belongs to player 0, the others to player 1. The parameter is tested to be divisible by 4 (depending on the inner functions used, sometimes it should even be divisible by 8 or 16).

Return value

The adequate description of the game to be used as the input arguments of the function computing the strategy.

Environment

Package

This function belongs to the package SIA-GENERATOR.

Implementation

There are inner functions `c1,..,c3,d1,..c3` used to determine connections between vertices. There are several possibilities implemented (but only one is uncommented). Of course new implementations of these functions can be added. The function itself checks for and ignores edges defined multiple times.

The appropriate macros to define parity games are used.

Bugs / Errors

See also

`gen-example6, gen-example6c, c-parity-1arg`

Name

`gen-example6c` is a variant of `gen-example6b`, which is restricted to make the source code easier readable.

Synopsis

`(gen-example6c` *`gamesize`*`)`

Description

This function differs from `gen-example6b` only in the way that the functions, which are used to define the edges, are restricted to the ones, which were used in the paper [BLV96].

Parameter

`gamesize` This parameter gives the number of vertices of the parity graph. Half of it belongs to player 0, the others to player 1. The parameter is tested to be divisible by 4.

Return value

The adequate description of the game to be used as the input arguments of the function computing the strategy.

Environment

Package

This function belongs to the package SIA-GENERATOR.

Implementation

This implementation does **not** check for edges defined multiple times any more and works only with the functions as defined in the paper.

The appropriate macros to define parity games are used.

Bugs / Errors

See also

`gen-example6, gen-example6b, c-parity-1arg`

Name

`gen-example7` creates the example given in [SV00] to introduce the user interface.

Synopsis

`(gen-example7` $\boldsymbol{n}$`)`

Description

This simple, scalable example graph (bipartite) consists of $\boldsymbol{n}$ groups each consisting of two vertices, which are connected in both directions. These groups are connected in order of increasing vertex label by an edge. The vertices 1 to $2n-1$ are colored 1, the vertex $2n$ is colored 0. So, player 0 wins from all vertices.

Parameter

$\boldsymbol{n}$ Factor to scale the graph, which gives the number of groups.

Return value

The adequate description of the game to be used as the input arguments of the function computing the strategy.

Environment

Package

This function belongs to the package SIA-GENERATOR.

Implementation

The appropriate macros to define parity games are **not** used.

Bugs / Errors

See also

`gen-example8`, `gen-example9`, `c-parity-1arg`

Name

`gen-example8` creates the same example as `gen-example7`. This time the appropriate macros are used.

Synopsis

`(gen-example8` n`)`

Description

This simple, scalable example graph (bipartite) consists of n groups each consisting of two vertices, which are connected in both directions. These groups are connected in order of increasing vertex label by an edge. The vertices 1 to $2n-1$ are colored 1, the vertex $2n$ is colored 0. So, player 0 wins from all vertices.

Parameter

n Factor to scale the graph, which gives the number of groups.

Return value

The adequate description of the game to be used as the input arguments of the function computing the strategy.

Environment

Package

This function belongs to the package SIA-GENERATOR.

Implementation

The appropriate macros to define parity games are used.

Bugs / Errors

See also

`gen-example7, gen-example9, c-parity-1arg`

Name

`gen-example9` creates the same example as `gen-example7` and `gen-example8`.

Synopsis

`(gen-example9` *n*`)`

Description

This simple, scalable example graph (bipartite) consists of *n* groups each consisting of two vertices, which are connected in both directions. These groups are connected in order of increasing vertex label by an edge. The vertices 1 to $2n-1$ are colored 1, the vertex $2n$ is colored 0. So, player 0 wins from all vertices.

The difference to the definition by `gen-example8` is the use of abbreviations to simulate infix notation in arithmetic terms. This is thought of being easier to read for users not familiar with Lisp.

Parameter

n Factor to scale the graph, which gives the number of groups.

Return value

The adequate description of the game to be used as the input arguments of the function computing the strategy.

Environment

Package

This function belongs to the package SIA-GENERATOR.

Implementation

The appropriate macros to define parity games and abbreviations are used.

Bugs / Errors

See also

`gen-example7`, `gen-example8`, `c-parity-1arg`

Name

`gen-example10` creates the example indeed published in [Jur00].

Synopsis

(gen-example10 *levels blocks*)

Description

This function describes the version of the graph, indeed mentioned in the given paper. The definition uses the macros to define the game. There are various connections between levels and blocks, which are described in the paper.

Parameter

levels Factor to scale the graph, which gives the number of horizontal copies of a line (consisting of blocks).

blocks The number of vertical copies of a block, which is the smallest unit.

Return value

The adequate description of the game to be used as the input arguments of the function computing the strategy.

Environment

Package

This function belongs to the package SIA-GENERATOR.

Implementation

The appropriate macros to define parity games are used.

Bugs / Errors

See also

`gen-example1`, `gen-example1-b`, `gen-example1b`, `gen-example1c`, `c-parity-1arg`

2.2.6 Package SIA-BASIS

This package contains functions and macros with basic functionality, which are useful for several packages and even more applications. For a more detailed description we refer the user to the particular function.

A short overview is given here:

- Functions/macros to work with lists.
- Environment to use abbreviations.
- An alphabet structure giving numbers to symbols.
- Transformation operations between pair-lists, association-lists and hash-tables.
- Function to simplify a description of a transition system.

Name

`imapcar` applies the given function to each element of the given lists and the position of the elements in their lists.

Synopsis

(imapcar *fn* *first-parameter* [*rest-parameter-list*])

Description

This macro works similarly to the Lisp-function `mapcar`, but with the additional feature, that the position of an entry is the first argument to the given function.

Therefore the function *fn* must have one parameter more than there are lists in the function call. This function is then applied to the position of the list element and the corresponding list elements in all given lists.

The result list is as long as the shortest list given.

Parameter

fn Function with one more argument than there are lists, which returns the entry of the resulting list.

first-parameter At least one list has to be given to this macro.

rest-parameter-list Further lists are allowed, but the function must be able to cope with them.

Return value

A new list built from the given lists by applying the function to the list-entries and the position in the list. The length of the shortest given list determines the length of the resulting list.

Environment

Package

This macro belongs to the package SIA-BASIS.

Implementation

Bugs / Errors

See also

Name

`rpush` adds a new element to the end of a list.

Synopsis

(rpush *value place*)

Description

This macro corresponds to the Lisp macro `push`, which adds an element at the beginning of a given list. `rpush` does the same, only at the end of the list.

Parameter

value The new value to put into the list.

place The list, the new value is added to.

Return value

The new, changed list is returned, but more important is the side-effect that *place* is changed.

Environment

Package

This macro belongs to the package SIA-BASIS.

Implementation

Bugs / Errors

See also

Name

`abbreviations` is a synonym for the Lisp special form `symbol-macrolet`.

Synopsis

`(abbreviations` *`body`*`)`

Description

This special form is used to introduce names, which are expanded to the given expressions during execution. This is used to simulate infix notation by naming arithmetic terms in Lisp-normal prefix notation by the corresponding infix notation. Of course there are several additional possibilities to use this macro.

Parameter

`body` The body should consist of a list of pairs `(`*`name expression`*`)` and a normal body, where the names are valid. For more details visit the description of `symbol-macrolet`.

Return value

The return value of the inner body is handed over.

Environment

Package

This macro belongs to the package SIA-BASIS.

Implementation

Bugs / Errors

See also

`symbol-macrolet` in the description of Common Lisp.

Name

`integer-seq` creates a list of integers.

Synopsis

`(integer-seq` *m n*`)`

Description

This function simplifies the generation of integer lists. It creates a list starting with *m* and ending with *n*-1.

Parameter

m, n Integers, where $m < n$ otherwise the result is `NIL`.

Return value

A list of integers from *m* to *n*-1.

Environment

Package

This function belongs to the package SIA-BASIS.

Implementation

Bugs / Errors

See also

Name

`mapatoms` recursively applies the function `fn` to every atomic element.

Synopsis

(mapatoms *fn str* [*is-atom*])

Description

The function can be applied to lists or vectors (also containing lists and vectors as elements). It returns the corresponding list or vector where each atom is replaced by the result of the function call *fn* with the atomic element as the argument. As an optional argument a function saying what is an atom can be given, so that even lists and vectors can be treated as 'atomic'.

Parameter

fn The function which shall be applied to all atomic entries of *str*.

str The initial list or vector or what else the function is applied to.

is-atom This optional argument has to be a function saying `NIL` when the argument is not supposed to be atomic and something else if it is supposed to be atomic.

Return value

The corresponding list or vector to *str* where each atom is replaced by the result of the function call *fn* with the atomic element as the argument.

Environment

Package

This function belongs to the package SIA-BASIS.

Implementation

Anything else but vectors and lists and the entries for which *is-atom* returns not `NIL` is treated as being atomic.

Bugs / Errors

See also

Name

`make-alphabet` creates the alphabet structure for the given list of symbols if possible.

Synopsis

`(make-alphabet` *`symbols`*`)`

Description

The symbols are used to create a new alphabet. An error occurs if there are symbols which can not be differenced. Otherwise the alphabet structure is returned.

Parameter

`symbols` A list of symbols (unique!) which shall be used as an alphabet.

Return value

The alphabet structure for the given list of symbols.

Environment

Package

This function belongs to the package SIA-BASIS.

Implementation

Bugs / Errors

See also

`nth-symbol, symbol-no, alphabet-size`

Name

`nth-symbol` returns the symbol of the *n*-th entry of the alphabet.

Synopsis

`(nth-symbol` *`alphabet n`*`)`

Description

Everything is said already.

Parameter

alphabet An alphabet defined with `make-alphabet`.

n The number of a symbol in the `alphabet`.

Return value

The symbol corresponding to the given number.

Environment

Package

This function belongs to the package SIA-BASIS.

Implementation

Bugs / Errors

See also

`make-alphabet, symbol-no`

Name

`symbol-no` returns the number of the given *symbol*, if it belongs to the given *alphabet*.

Synopsis

`(symbol-no` *alphabet symbol*`)`

Description

The symbol is searched in the alphabet structure and the corresponding number is returned, if the element is found.

Parameter

alphabet An alphabet defined with `make-alphabet`.

symbol A symbol belonging to `alphabet`.

Return value

The number of the symbol, if it belongs to the alphabet.

Environment

Package

This function belongs to the package SIA-BASIS.

Implementation

Bugs / Errors

See also

`make-alphabet, nth-symbol`

Name

`alphabet-size` returns the number of entries in the given instance of the alphabet structure.

Synopsis

(alphabet-size *alphabet*)

Description

Nothing more to say.

Parameter

alphabet An alphabet defined with `make-alphabet`.

Return value

The size of the alphabet.

Environment

Package

This function belongs to the package SIA-BASIS.

Implementation

Bugs / Errors

See also

`make-alphabet`

Name

`pairs-to-adjacents-vector` transforms a list of pairs to an adjacents-vector of the given size.

Synopsis

`(pairs-to-adjacents-vector size transitions)`

Description

This function creates a new vector with size *size* and fills it with the elements of the pair list, whereby the first element of a pair is used as the index of the vector and should therefore be in a suitable range.

Parameter

size The size of the resulting vector.

transitions A list of pairs where at least the first element of a pair is a number between 0 and $size - 1$.

Return value

A vector containing the information of the pair list as an adjacents vector with size *size*.

Environment

Package

This function belongs to the package SIA-BASIS.

Implementation

Bugs / Errors

See also

`pair-to-assoc`

Name

`assoc-to-pair` changes the data structure type from association-list to pair-list.

Synopsis

`(assoc-to-pair` *alist*`)`

Description

This function creates from each entry of the input association-list a list of pairs consisting of the key and a value for all values in the association lists and concatenates all the lists (for the keys) to one list together.

Parameter

alist An association list

Return value

A list of pairs, which contains the same information as the association-list.

Environment

Package

This function belongs to the package SIA-BASIS.

Implementation

Bugs / Errors

See also

`pair-to-assoc`, `hash-to-alist`

Name

`pair-to-assoc` changes the data structure type from pair-list to association-list.

Synopsis

`(pair-to-assoc` *`pairlist`*`)`

Description

This function gathers all the pairs of the input list which have the same key and puts them as an entry with the key argument and a list of values in the new list. Of course this is done for each key and thereby an association list is created.

Parameter

`pairlist` A list of pairs (multiple apprearance of keys or values are admitted).

Return value

A list, which is the corresponding association-list.

Environment

Package

This function belongs to the package SIA-BASIS.

Implementation

Bugs / Errors

See also

`assoc-to-pair, hash-to-alist`

Name

`hash-to-alist` constructs an association list from a hash table.

Synopsis

`(hash-to-alist` *h* `)`

Description

This function gets a hash table as an input and returns the corresponding association list by putting each pair key, value of the hash table in a list.

Parameter

h A hash table.

Return value

An association list of the hash table datas.

Environment

Package

This function belongs to the package SIA-BASIS.

Implementation

Bugs / Errors

See also

`pair-to-assoc, assoc-to-pair`

Name

`crossproduct` composes two lists to a new list.

Synopsis

(crossproduct *x y*)

Description

This function creates a new list from the two input lists, where an entry in the new list is a list with an element of the first list and an element of the second list for all pairs possible.

Parameter

x, y Lisp lists

Return value

A list representing the crossproduct of the two lists *x, y*.

Environment

Package

This function belongs to the package SIA-BASIS.

Implementation

Bugs / Errors

See also

Name

`simplify-graph` creates an association list for a graph given as a hash-table and replaces the names of the vertices by numbers.

Synopsis

`(simplify-graph` *`graph`*`)`

Description

This function simplifies the given graph by replacing maybe big(long) names by numbers. The graph is expected to have named transitions. These names are left unchanged.

Additionally the graph is converted from a hash-table to an association-list.

Parameter

`graph` A graph given as a hash-table, where the value to a key lists at first the label of the transition and then the vertices the key vertex is adjacent to.

Return value

An association list describing an equivalent graph where all names of vertices are given by numbers.

Environment

Package

This function belongs to the package SIA-BASIS.

Implementation

Bugs / Errors

See also

`ccs-gen`

2.2.7 Package SIA-MU-CALCULUS

This package contains functions to decide in which states of a given Kripke structure a modal μ-calculus formular holds. This is done, of course, by converting the problem into a parity game and solving this parity game. There are at least two versions of this transformation.

Furthermore a converter for graphs described in TRUTH terms is given, which returns the graph in a format used in this implementation.

To express μ-calculus formulars the following keywords are allowed, additionally to `and` and `or`: `mu`, `nu`, `box`, and `diamond`. The use of `not` is restricted within this calculus. See the descriptions of the functions for further information.

An important intern variable for the default behaviour of this package is the variable `make-game-fkt-default`. This variable specifies the function, which is used in the functions `eval-form2`, `eval-form3`, and `run-form3`, to construct the parity-game, if there is no function given explicitly. Default up to now is the function `make-game2`, another possibility is the function `make-game-fast`. Within this last function it is interesting which function is used to build up the formular-graph. Up to now this is the one, constructing a maybe smaller, simplified formular-graph. The functions above are only exported to give the user the choice between the two functions.

Name

`eval-form2` evaluates a "nice" formular on a given transition system.

Synopsis

(eval-form2 *transitions formular* [*make-game-fkt*
[:initial-strategy *initial-strategy*]])

Description

This function returns the states of the transition system in which the complete formular holds. This is done by solving the corresponding parity game and collecting the first components of the compounded states of the parity game, where the second component is the whole formular.

By using this function call it has to be assured that all names of variables are unique and `not` is not used in the formular, otherwise the function `eval-form3` would be more appropriate. This is meant with "nice" in the description above.

For further information on the run of the algorithm use `run-form3`.

Parameter

transitions The Kripke structure with labeled edges, for which the formular has to be checked.

formular A modal μ-calculus formular in which all names of variables are unique and `not` is not used.

make-game-fkt With this optional parameter a function to construct the parity game is given. Possible are up to now only `make-game2` or `make-game-fast`.

initial-strategy This variable specifies a function as described in the package SIA-STRATEGY to change the inital behaviour of the algorithm.

Return value

A list of states of the transition system in which the formular holds (or `NIL`).

Environment

The setting of the variable `make-game-fkt` as described in the introduction to this package.

Package

This function belongs to the package SIA-MU-CALCULUS.

Implementation

Bugs / Errors

See also

`eval-form3, run-form3`

Name

`eval-form3` works exactly like `eval-form2`, but replaces all variables by unique names and expands all `not`'s.

Synopsis

(eval-form3 *transitions formular* [*make-game-fkt*
    [:initial-strategy *initial-strategy*]])

Description

The only difference of this function against the function `eval-form2` is that all variables are replaced by unique names, so that the names need not be unique any more. Furthermore the `not`'s are expanded before the formular is evaluated. But still the formular has to be monotonic, because the expansion is simply done by a dual-operation.

Parameter

transitions The Kripke structure with labeled edges, for which the formular has to be checked.

formular A modal μ-calculus formular (monotonic).

make-game-fkt With this optional parameter a function to construct the parity game is given. Possible are up to now only `make-game2` or `make-game-fast`.

initial-strategy This variable specifies a function as described in the package SIA-STRATEGY to change the inital behaviour of the algorithm.

Return value

A list of states of the transition system in which the formular holds (or `NIL`).

Environment

The setting of the variable `make-game-fkt` as described in the introduction to this package.

Package

This function belongs to the package SIA-MU-CALCULUS.

Implementation

Bugs / Errors

See also

`eval-form2, run-form3`

Name

`run-form3` works exactly like `eval-form3`, but returns also information on the run of the algorithm.

Synopsis

(run-form3 *transitions formular* [*make-game-fkt*
[:initial-strategy *initial-strategy*]])

Description

The only difference of this function against the function `eval-form3` is that additional information about the run of the algorithm and the running-time is returned. Therefore the use of `not`'s is (in the restricted sense) allowed in the formular.

Besides the two new return parameters about the number of iterations and the real running-time of the algorithm, there is an output of the `time` function for the complete call of the function `eval-form`.

Parameter

transitions The Kripke structure with labeled edges, for which the formular has to be checked.

formular A modal μ-calculus formular (monotonic).

make-game-fkt With this optional parameter a function to construct the parity game is given. Possible are up to now only `make-game2` or `make-game-fast`.

initial-strategy This variable specifies a function as described in the package SIA-STRATEGY to change the inital behaviour of the algorithm.

Return value

Multiple values are returned:
- A list of states of the transition system in which the formular holds (or `NIL`).
- The number of iterations the C-program needed.
- The real-time the C-program needed.

Environment

The setting of the variable `make-game-fkt` as described in the introduction to this package.

Package

This function belongs to the package SIA-MU-CALCULUS.

Implementation

Bugs / Errors

See also

`eval-form2, eval-form3`

Name

`make-game-fast` constructs a parity game from a transition system and a μ-calculus-formular.

Synopsis

`(make-game-fast` *`transitions formular`*`)`

Description

This function is only exported to give the user the choice which function is used to build up the parity game.

Parameter

`transitions` The transition system.

`formular` The modal μ-calculus formular without `not`'s and with unique names for variables.

Return value

The information for a parity game as needed by `c-parity`.

Environment

Package

This function belongs to the package SIA-MU-CALCULUS.

Implementation

Within this function it is hidden, whether a small, simplified formular graph is constructed or not. See the function call for the construction of the formular graph for details.

Bugs / Errors

See also

`eval-form2, eval-form3, run-form3, make-game2`

Name

`make-game2` constructs a parity game from a transition system and a μ-calculus-formular.

Synopsis

(make-game2 *transitions formular*)

Description

This function is only exported to give the user the choice which function is used to build up the parity game.

Parameter

transitions The transition system.

formular The modal μ-calculus formular without `not`'s and with unique names for variables.

Return value

The information for a parity game as needed by `c-parity`.

Environment

Package

This function belongs to the package SIA-MU-CALCULUS.

Implementation

Bugs / Errors

See also

`eval-form2, eval-form3, run-form3, make-game-fast`

Name

`truth-converter` reads the description of a graph in TRUTH terms from files and returns a description as used in this implementation.

Synopsis

`(truth-converter` *`states-filename transitions-filename`*`)`

Description

This function is able to read files, which are created with TRUTH and describe a graph, and creates a graph description in "our" terms, which is returned as well as an alphabet, which gives the assignment between numbers and the names of the states.

Parameter

`states-filename` The filename of the file with the information about the states.

`transitions-filename` The filename of the file with the information about the transitions.

Return value

Multiple values are returned:
- The graph as a vector of transition lists (each possibly consisting of serveral lists with different labels) where numbers are used instead of the names of the states.
- An alphabet for the assignment between the numbers and the states.

Environment

Package

This function belongs to the package SIA-MU-CALCULUS.

Implementation

Bugs / Errors

See also

Appendix A

Installation

In this part additional information how to use this toolkit is given. The topics range from implementation details to the directory structure which is expected.

Files of the toolkit

As already said the algorithm is written in C to run efficiently, whereas the user interface requires a Lisp interpreter. The algorithm is implemented within only one C-source file named `c-parity-obda.c`, therefore the executive version is assumed to have got the name `c-parity-obda`.

The user interface in Lisp is divided into six packages and so there are six files (one for each package) and a central one, which helps working with all the packages. In the following list the version numbers are only examples of course, which reflect the actual state. In fact it is not necessary to delete older versions, if always the central file `omega-X.X.X.lsp` is loaded. This loads the newest versions of all the other files. Therefore to use this toolkit a user has to put the following files into the same directory and has to make this directory the working directory of the Lisp interpreter:

- `c-parity-obda`
- `omega-0.1.1.lsp`
- `ccs-0.0.4.lsp`
- `sia-basisfcts-0.2.8.lsp`
- `sia-generators-0.2.9.lsp`
- `sia-mu-calculus-0.2.15.lsp`
- `sia-parity-graph-0.2.9.lsp`
- `sia-strategy-0.2.10.lsp`

Running the toolkit

To run the user interface a Lisp interpreter is needed, which understands the Lisp language conform to [Ste84], [Ste90] (older parts) as the one we used `clisp`.

The working directory of the Lisp interpreter has to be the directory with all the files mentioned above in it. The next step is to load the file `omega-0.1.1.lsp` or a newer version of this file into the interpreter. This causes all the other files (except the C-program) to be loaded as well, so that all the other packages are available.

Then simply the appropriate functions can be used to work with the toolkit. The package `SIA-STRATEGY` expects the C-program also in the working directory.

Maybe it is important to mention that the exchange between the Lisp user interface and the C-program works by temporary files (hidden), so that the user needs write access permission to the directory to run the toolkit.

Futher results of a run of the algorithm

There is further information on the run of the algorithm returned not only that what the user gets in the Lisp interpreter. The file `auswertungsbild` contains the improvement trace (see [SV00] for explanation) and the file `auswertung` other - hopefully self-explaining - information on the run of the algorithm.

Bibliography

[BLV96] Nils Buhrke, Helmut Lescow, and Jens Vöge. Strategy construction in infinite games with Streett and Rabin chain winning conditions. In T. Magaria and B. Steffen, editors, *TACAS'96*, volume 1055 of *LNCS*, pages 207–225. Springer, 1996.

[EJ91] E. Allen Emerson and Charanjit S. Jutla. Tree automata, mu-calculus and determinacy. In *Proc. 32nd IEEE Symp. on the Foundations of Computing*, pages 368 – 377, 1991.

[EJS93] E. Allen Emerson, Charanjit S. Jutla, and A. Prasad Sistla. On model checking for fragments of μ-calculus. In C. Courcoubetis, editor, *Computer Aided Verification*, number 697 in Lecture Notes in Computer Science, pages 385–396, Berlin, 1993. Springer Verlag.

[Jur00] Marcin Jurdziński. Small progress measures for solving parity games. In Horst Reichel and Sophie Tison, editors, *STACS 2000, 17th Annual Symposium on Theoretical Aspects of Computer Science, Proceedings*, volume 1770 of *LNCS*, pages 290–301, Lille, France, February 2000. Springer.

[McN93] Robert McNaughton. Infinite games played on finite graphs. *Ann. Pure Appl Logic*, 65:149 – 184, 1993.

[Pur95] Anuj Puri. *Theory of hybrid systems and discrete event systems.* Ph.d. thesis, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, December 1995. Memorandum No. UCB/ERL M95/113.

[Ste84] Guy L. Steele. *Common Lisp - The Language, 1st edition.* Digital Press, 1984.

[Ste90] Guy L. Steele. *Common Lisp - The Language, 2nd edition.* Digital Press, 1990.

[SV00] Dominik Schmitz and Jens Vöge. Implementation of a strategy improvement algorithm for finite-state parity games. In *Conference on Implementation and Application of Automata (CIAA)*, 2000.

[Tho97] Wolfgang Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Language Theory*, volume III, pages 389–455. Springer-Verlag, New York, 1997.

[VJ00] Jens Vöge and Marcin Jurdziński. A discrete strategy improvement algorithm for solving parity games. In *Computer Aided Verification (CAV)*, 2000.

Index