A compact file format for labeled transition systems

I.A. van Langevelde

Software Engineering (SEN)

SEN-R0102 January 31, 2001

Report SEN-R0102
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

A Compact File Format for Labeled Transition Systems

Izak van Langevelde
email: izak@cwi.nl

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT
A compact open file format for labeled transition systems, which are commonly used in specification and verification of concurrent systems, is introduced. This combination of openness, both in specification and implementation, and compactness is unprecedented, since existing formats in this field are either not compact, yielding files too large to be easily handled, or proprietary, hampering the development of efficient tools. Therefore, the development of this SVC format was initiated to facilitate the development of state-of-the-art tools for the analysis of concurrent systems and communication protocols.

The SVC format is specified by its binary layout and the underlying compression scheme, based on Lempel-Ziv and dynamic Huffman encoding. Also, an application programming interface is defined. The compression scheme is compared with existing algorithms.

2000 Mathematics Subject Classification: 68N30 Specification and verification; 68P30 Coding and information theory
1998 ACM Computing Classification System: D.2.1 Requirements/Specifications; D.2.4 Program verification; E.4 Coding and information theory
Keywords and Phrases: labelled transition systems, file format, compression
Note: The research reported here is part of the Systems Validation Centre (SVC)

1. Introduction

The most influential approach in the field of automated analysis of concurrent system is *model checking* [3], which consists of generating a Kripke model of the system and checking whether the model satisfies the logical formulae that denote the desired properties. The bottleneck here is the generation of the model, which typically contains millions of states and transitions between these and takes up several gigabytes of disk space. For instance, the *instantiator* tool from the μCRL tool set [9] typically generates files of several gigabytes in days. The current study targets the efficient storage of these models, or *labeled transition systems* as these will be referred to.

The importance of a compact file format has been recognised before: the CÆSAR/ALDÉBARAN tool set [4] incorporates the BCG format (*Binary Coded Graphs*) which offers an efficient implementation of an extremely compact file format. The BCG format, however, has two major flaws: 1) its specification and implementation are not public; 2) it is limited to states being labelled by natural numbers. If it were not for the first flaw, it might be feasible to tackle the second by improving on the existing BCG format, possibly ensuring downward compatibility in the sense that new software may read old files, but as it is, BCG is a black box that is only fully understood by its creators.

The current document proposes a new file format for the compact representation of transition systems that improves on the BCG format at exactly these two points. The new file format, dubbed SVC after the *Systems Validation Centre Project*[1] which supported its development, is open in its specification and implementation, and it allows states being labeled by arbitrary terms[2].

[1]The Systems Validations Centre is a cooperative effort of the Centre for Telematics and Information Technology of the Twente University, CWI and the Telematics Institute

[2]This report describes SVC version 1.2

This paper is organised as follows. Section 2 gives an overview of the file format. Section 3 specifies the file format and the algorithm used to compress SVC files. Section 4 compares the SVC format with existing compression schemes and Section 5 outlines future research. The SVC application programming interface is given in Appendix I.

2. Overview

A labeled transition system is basically a directed graph, of which the nodes are called *states* and the directed edges are *transitions*. One of the states is designated *initial*. Edges are labeled by *actions*, which denote the action that brings the system from one state into another. Although this simple notion of labeled transition systems covers the vast majority of existing applications, two trends are worth to be anticipated. First, classical applications of labeled transition systems, for instance process algebra [5], focus on the actions that can be executed in a state; more recent applications shift focus from system behaviour to the internals of the system, in terms of program variables, stacks and program counters. Second, recent variants of the concept rely on additional parameters being defined for transitions. An example is formed by *stochastic systems*, which associate with each transition the λ parameter from an exponential distribution. Although the vast majority of algorithms and implementations currently available still adheres to the classical action-based view on labeled transition systems, these trends are anticipated by allowing states to contain information and transitions to have a *parameter*.

A labeled transition system as stored in an SVC file consists of states and transitions, each of which connects one state to another and has an action and a parameter associated with it. States, actions and parameters are all represented by terms as defined in [2], that is, 'classical terms' like $f(a, g(b, c))$, strings like "monkeys and bananas", integers and real numbers, and lists of these, like ["one", 2, $add(2, 1)$].

Apart from the actual labeled transition system there is *meta information* about this system, such as the numbers of states, transitions and actions, the program that created it and comments. Some of this meta information is essentially read-only, for instance the number of states, while other information can also be written by the user. This meta information is stored in the *header* of an SVC file:

- *comments* – description of the file content
- *creator* – the program that created the file
- *date* – file creation date (read-only)
- *file name*– file name (read-only)
- *type* – transition system type
- *version* – file version
- *# labels* – number of distinct action labels (read-only)
- *# states* – number of distinct states (read-only)
- *# parameters*– number of distinct transition parameters (read-only)
- *# transitions*– number of transitions (read-only)

Finally, future changes in the definition of the file format are anticipated by a simple mechanism for version control: any implementation is able to check whether it will be able to read a given SVC file or not, by checking the *format version number* of the file. Finally, an SVC file contains a CRC32 [14] checksum, making it possible to check its integrity.

3. The file layout

Svc is a binary format, that can be considered as a large data structure which is hierarchically composed from, at its lowest level, atomic data types. The svc data structures are (de)composed by the application programming interface, that will be presented in Appendix I, while the (de)composition of the atomic data types is left to the user of this interface.

3.1 The atomic data types

The atomic data types are strings, integers and terms. The latter is defined in the style of *Annoted Terms* (ATerms [2]), to include function applications, strings, numbers and lists. Each of these comes in two flavours: compressed and uncompressed.

These data types are specified in terms of bits and bytes, as specified in the following BNF syntax. In this syntax, representations of ascii characters are denoted by the character they represent in single quotes. As only printable characters are used, the most significant bit is dropped so that characters are stored in seven bits.

Integers are represented by at most four bytes, big endian, preceded by one sign bit and two length bits (see Figure 1). The number 0 is represented by "1 00 00000000". The bit sequence "0 00 00000000" is a special marker for "the illegal integer", which will be used as a delimiter.

```
⟨string uncompressed⟩   ::= ⟨character⟩* 0000000
⟨int uncompressed⟩      ::= ⟨bit⟩00⟨byte⟩|
                            ⟨bit⟩01⟨byte⟩⟨byte⟩ |
                            ⟨bit⟩10⟨byte⟩⟨byte⟩⟨byte⟩ |
                            ⟨bit⟩11⟨byte⟩⟨byte⟩⟨byte⟩⟨byte⟩
⟨term uncompressed⟩     ::= ⟨aterm⟩ 0000000
⟨aterm⟩                 ::= ⟨application⟩ | ⟨number⟩ | ⟨list⟩ | ⟨string⟩
⟨application⟩           ::= ⟨identifier⟩ | ⟨identifier⟩ '(' ⟨aterm-list⟩ ')'
⟨number⟩                ::= ⟨integer⟩ | ⟨real⟩
⟨aterm-list⟩            ::= ⟨aterm⟩ (',' ⟨aterm⟩)*
⟨integer⟩               ::= ⟨decimal-integer⟩
⟨real⟩                  ::= ⟨decimal-real⟩
⟨identifier⟩            ::= ⟨letter⟩(⟨letter⟩|⟨decimal-digit⟩)*
⟨decimal-integer⟩       ::= [⟨sign⟩] ⟨decimal-digit⟩+
⟨decimal-real⟩          ::= [⟨sign⟩] ⟨decimal-digit⟩* '.' ⟨decimal-digit⟩+
⟨sign⟩                  ::= '–' | '+'
⟨list⟩                  ::= '[' ⟨aterm-list⟩ ']' | '['']'
⟨string⟩                ::= '"' ⟨character⟩* '"'
⟨letter⟩                ::= 'A' | ... | 'Z' | 'a' | ... | 'z'
⟨decimal-digit⟩         ::= '0' | ... | '9'
⟨int compressed⟩        ::= ⟨bit⟩+
⟨aterm⟩                 ::= ⟨bit⟩ +
⟨string compressed⟩     ::= ⟨bit⟩ +
```

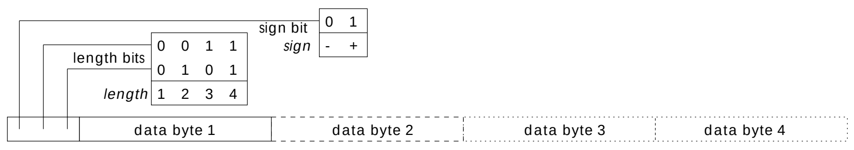

Figure 1: integer

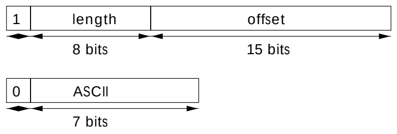

Figure 2: Lempel-Ziv token

⟨bit⟩	::=	0 | 1
⟨byte⟩	::=	⟨bit⟩⟨bit⟩⟨bit⟩⟨bit⟩⟨bit⟩⟨bit⟩⟨bit⟩⟨bit⟩
⟨character⟩	::=	⟨bit⟩⟨bit⟩⟨bit⟩⟨bit⟩⟨bit⟩⟨bit⟩⟨bit⟩
⟨the invalid integer⟩	::=	10000000000
⟨the invalid term⟩	::=	‘”’‘E’‘S’‘C’‘”’‘(’‘”’‘N’‘I’‘L’‘”’‘)’ 0000000

According to this syntax, compressed data is stored as arbitrary length sequences of bits. The underlying reason is the use of Huffman coding, which uses no fixed code length, but assigns shorter codes to less frequent data elements. The compression scheme will be described in Section 3.3.

3.2 data structures

An SVC file consists of five parts, the *index flag*, the *index*, the *version header*, the *header*, the *body* and *the trailer*. The index flag specifies whether the transition system is limited to states being natural numbers, or whether states can be assigned arbitrary terms. The index contains the position in bytes of the other components and the version header contains the format version number of the file. The specification of these two components will remain unchanged to guarantee that any implementation of any future revision of the SVC format is able to check for any file in any version of the SVC format whether it can read this file. The header contains the meta information on the file, the body contains the transitions, terminated by a special marker and the trailer contains the CRC32 checksum.

⟨svcfile⟩	::=	⟨svcfile indexed⟩ | ⟨svcfile non-indexed⟩
⟨svcfile indexed⟩	::=	1⟨index⟩⟨version header⟩⟨body indexed⟩⟨header⟩⟨trailer⟩
⟨svcfile non-indexed⟩	::=	0⟨index⟩⟨version header⟩⟨body non-indexed⟩⟨header⟩⟨trailer⟩
⟨index⟩	::=	⟨header position⟩⟨body position⟩⟨trailer position⟩⟨version position⟩
⟨header position⟩	::=	⟨int uncompressed⟩
⟨body position⟩	::=	⟨int uncompressed⟩
⟨trailer position⟩	::=	⟨int uncompressed⟩
⟨version position⟩	::=	⟨int uncompressed⟩
⟨body indexed⟩	::=	⟨transition indexed⟩* ⟨end of body indexed⟩
⟨body non-indexed⟩	::=	⟨transition non-indexed⟩* ⟨end of nody non-indexed⟩
⟨end of body non-indexed⟩	::=	⟨the invalid term⟩
⟨end of body indexed⟩	::=	⟨the invalid integer⟩
⟨transition indexed⟩	::=	⟨state indexed⟩⟨label⟩⟨state indexed⟩⟨parameter⟩
⟨transition non-indexed⟩	::=	⟨state non-indexed⟩⟨label⟩⟨state non-indexed⟩⟨parameter⟩
⟨state indexed⟩	::=	⟨integer compressed⟩
⟨state non-indexed⟩	::=	⟨term compressed⟩
⟨label⟩	::=	⟨term compressed⟩
⟨parameter⟩	::=	⟨term compressed⟩
⟨header⟩	::=	⟨filename⟩
		⟨creation date⟩
		⟨file version⟩

```
                             ⟨file subtype⟩
                             ⟨file creator⟩
                             ⟨number of states⟩
                             ⟨number of transitions⟩
                             ⟨number of labels⟩
                             ⟨number of parameters⟩
                             ⟨initial state⟩
                             ⟨comments⟩
⟨filename⟩               ::= ⟨string uncompressed⟩
⟨creation date⟩          ::= ⟨string uncompressed⟩
⟨file version⟩           ::= ⟨string uncompressed⟩
⟨file subtype⟩           ::= ⟨string uncompressed⟩
⟨file creator⟩           ::= ⟨string uncompressed⟩
⟨number of states⟩       ::= ⟨integer uncompressed⟩
⟨number of transitions⟩  ::= ⟨integer uncompressed⟩
⟨number of labels⟩       ::= ⟨integer uncompressed⟩
⟨number of parameters⟩   ::= ⟨integer uncompressed⟩
⟨initial state⟩          ::= ⟨string uncompressed⟩
⟨comments⟩               ::= ⟨string uncompressed⟩
⟨trailer⟩                ::= ⟨int uncompressed⟩
```

3.3 Compression scheme

The compression scheme used in SVC files is based on a number of assumptions:

1. There is good deal of overlapping among state labels, in the sense that substrings found in one label are likely to occur in other labels
2. There is a good deal of overlapping among actions
3. There is a good deal of overlapping among transition parameters
4. There is a good deal of regularity in the occurring of integer state labels in the sequence of transitions
5. Some action labels occur more often than others

Assumptions 1, 2 and 3 suggest the use of a dictionary-based compression scheme, where strings are compressed by representing these by a reference into a dictionary. Assumption 4 and 5 suggest the use of a statistical method, where strings with a high frequency of occurrence are assigned shorter codes than strings with a low frequency. The SVC compression scheme is a combination of both.

Terms are compressed by a combination of dynamic Huffman encoding [11] and a Lempel-Ziv variant based on LZSS [15]. That is, each first occurrence of a substring is compressed by this LZSS variant, preceded by an escape sequence, and assigned a code. Each other occurrence of this substring is then represented by its code, which is subject to change according to the dynamic Huffman algorithm which assures that frequent codes are shorter than less frequent ones.

The Lempel-Ziv variant is based on LZSS [15] with the one deviation that the search buffer is stored in the same circular queue as the look-ahead buffer, instead of storing it in a binary search tree. The tokens consist of one flag bit followed by 8 length bits and 15 offset bits (flag = 1) or the 7 least significant bits of an ASCI character (flag=0) (see Figure 2). Terms are accumulated in the look-ahead buffer, making it possible to benefit from similarities across terms. Also, for states, action labels and parameters separate buffers are used.

Integers are first processed by *differencing* [8] and then compressed by the same dynamic Huffman algorithm that is used to compress strings.

	transition system	number of states	number of transitions	number of actions	average state	average action
1	alternating bit protocol	144	183	11	68	4
2	leader election protocol	236	584	2	263	3
3	sliding window protocol	319732	1936422	11	129	4
4	Firewire protocol	371804	641565	61	567	9
5	Splice system	505528	1403242	97	488	37

Table 1: The parameters of the labelled transition systems

The differencing scheme is as follows. The sequence of integers to be compressed is partitioned into two sequences, i.e. the sequence consisting of integers at odd positions and the sequence consisting of integers at even positions. Then, each integer is replaced by the difference with its predecessor in its sequence.

Definition 1 *Suppose $n_0, n_1, \cdots$ is a sequence of integers, then the corresponding differential sequence $d_0, d_1, \cdots$ is defined by:*

- $d_0 = n_0$
- $d_1 = n_1$
- $d_{2i} = n_{2i} - n_{2i-2}$
- $d_{2i+1} = n_{2i+1} - n_{2i-1}$

4. Comparison with existing work

There is one file format, known to the author of the current report, that directly competes with the SVC format: the BCG format. There are other file formats for labeled transition systems, for instance FC2 [12] and PSF [13] but these do not strive for compactness, the central feature of SVC. However, it is useful to compare the SVC compression scheme with other schemes, in order to assess the SVC compression rate.

The SVC compression is compared with the following compression schemes:

binary ATerm format (baf) Special-purpose format for storing terms, based on maximal sharing of subterms, i.e. each subterm is stored exactly once. The ATerm library [2] is available under the GNU General Public License [6].

binary coded graphs (BCG) File format for labelled transition systems, with states being limited to natural numbers. File format, compression scheme and implementation are proprietary [4]

GNU zip files (gzip) Popular general-purpose compression format, based on a combination of Lempel-Ziv and static Huffman. It is available under the GNU general public license [1]. For the comparison, GZIP was used at its optimal compression, i.e. it was run with the -BEST flag.

Aldébaran format (aut) Text-based format, with states limited to natural numbers [4]

The transition systems used in the comparison are given in Table 1. For each system the numbers of states, transitions and actions are given, and the average length in characters of the labels of states and actions. These averages are calculated as follows (n is the number of transitions):

average state length $= \frac{\sum_{s \xrightarrow{a} t} |s|+|t|}{2 \cdot n}$

average action length $= \frac{\sum_{s \xrightarrow{a} t} |a|}{n}$

	TEXT		GZIP		BAF		BCG		SVC	
system	KB	%	KB	%	KB	%	KB	%	KB	%
1	27573	100	1531	5.55	3521	12.8	–	–	1712	6.20
2	314195	100	4266	1.36	16295	5.19	–	–	3616	1.15
3	523055783	100	12538168	2.40	37424146	7.15	–	–	12296533	2.35
4	738596621	100	7045472	0.95	28346037	3.84	–	–	6999769	0.95
5	5728503398	100	57362770	1.00	–	-.–	–	–	24257108	0.42
average	1398099514	100	15390441	1.10	16447500	5.21	–	–	8711748	0.62

Table 2: Compression rates for non-indexed files

For each of the transition systems from Table 1 two variants were subjected to comparison. The non-indexed variant contains the full state vectors, represented as terms, and the indexed variant reduces each state vector to its index number. The former variant is represented in a plain text file, which is a straightforward generalisation of the the .aut format, and the latter variant is represented by a file in .aut format. These two formats form the base of the comparison, in the sense that compression rates are calculated with respect to these formats; by definition, the compression rate for these base formats is 100%. The results of the comparison are shown in Table 2 and 3. Note that BCG does not apply to the nonindexed variant; for the largest transition system no data on BAF are available, since generating the BAF file was not possible within the available memory .

The results show that for non-indexed files SVC compresses slightly better than GZIP while for indexed files SVC compresses slightly better than BCG. However, for indexed files SVC and BCG compress twice as good as GZIP.

It must be stressed that SVC, just as BCG, is more than a compression scheme: it allows transitions to be written and compressed one by one, without having to first generate the complete transition system on disk or in memory and, second, to compress the complete system. The only reason for comparing SVC to BAF and GZIP is to assess its compression rate: the actual use of these to compress systems would be impractical.

5. Conclusions and future work

The SVC compression scheme described in this report compares favourably with existing schemes, like GZIP for non-indexed files and BCG for indexed files. It is premature to claim that SVC is the better of the three after a comparison based on five examples; a claim like this requires a broader analysis. However, compression rate is just one of the parameters the file format can be evaluated on.

The original motivation for taking up the development of the SVC format was the need for an open specification and implementation, rooted in the conviction that openness is the key to scientific progress. An open file format may act as a catalyst for the further development of tools for the analysis

	AUT		GZIP		BAF		BCG		SVC	
system	KB	%	KB	%	KB	%	KB	%	KB	%
1	3008	100	868	28.9	2804	93.2	3853	128	353	11.7
2	8827	100	1641	18.6	6215	70.4	3492	39.6	715	8.10
3	44897785	100	8315124	18.5	30872097	68.7	6077961	13.5	5947667	13.2
4	17809563	100	2985805	16.8	13870681	77.9	1424226	8.00	926216	5.20
5	79257948	100	7421031	9.36	26303382	33.2	3526297	4.45	4048262	5.11
average	28395426	100	3744894	13.2	14211036	50.0	2207166	7.77	2184643	7.69

Table 3: Compression rates for indexed files

of concurrent systems, some of which complement each other, and some of which may improve on existing tools, possibly making them obsolete. As such, a common open format serves as a fertile common ground on which a coherent body of tools starts to grow, evolving towards more analytical power, broader coverage of features to be analysed and higher applicability.

The success of the SVC file format can only then be claimed if it indeed proves to stimulate the fruitful development it aims at. This is where evangelisation enters the stage. It must be demonstrated how the SVC format can be fruitfully used in the development of tools for the analysis of concurrent systems, and how other tools can benefit from interfacing with these tools. The format was incorporated in the μCRL tool set, and the developers of TIPPTOOL [10] have shown interest in the development of SVC, so there is a narrow basis for the use of this file format.

Secondary work should be aimed at the implementation of the SVC interface. The current prototype implementation uses a simple Lempel-Ziv implementation, in which the look-ahead buffer is searched sequentially; more efficient algorithms are known from literature. The same holds for the dynamic Huffman implementation. Also, it might be possible to refine the current compression scheme to achieve a better compression rate.

Third and final, it was claimed recently that BCG does facilitate data to be stored into states [7]. However, this functionality appears to be hidden at the lower levels of this file format as it is not available via the high level programming interface available in public. Despite an urgent request, no information on this feature has been made available. A full comparison between BCG and SVC still remains to be done.

Availability

The SVC library is available from `ftp://ftp.cwi.nl/pub/izak/svc`. Also available from this URL is a set of tools based on the format, including converters to/from .aut format, a bisimulation reduction tool and a confluence reduction tool. The implementation was tested on Unix (Solaris/Irix/Linux). The library is available under the *GNU General Public License* [6].

Acknowledgements

Jan Friso Groote took the first step towards the development of a compact open file format for labeled transition systems. He contributed a number of improvements to early versions of the programming interface. He and Sjouke Mauw commented upon this text, suggesting a number of improvements.

References

1. *The* GZIP *home page*. `http://www.gzip.org`.
2. BRAND, M. G. J. VAN DEN, JONG, H. A. DE, KLINT, P, AND OLIVIER, P. A. Efficient annotated terms. Tech. Rep. SEN-R0003, CWI, Mar. 2000. Available from `http://www.cwi.nl/ftp/CWIreports/SEN/SEN-R0003.ps.Z`.
3. CLARKE, E. W., GRUMBERG, O., AND PELED, D. A. *Model Checking*. The MIT Press, 1999.
4. FERNANDEZ, J.-C., GARAVEL, H., KERBRAT, A., MATEESCU, R., MOUNIER, L., AND SIGHIREANU, M. CADP (Cæsar/Aldébaran development package): A protocol validation and verification toolbox. In *Proceedings of the 8th Conference on Computer-Aided Verification* (Aug. 1996), R. Alur and T. A. Henzinger, Eds., vol. 1102 of *LNCS*, Springer Verlag, pp. 437–440.
5. FOKKINK, W. *Introduction to Process Algebra*. Texts in Theoretical Computer Science (an EATCS Series). Springer Verlag, Jan. 2000.
6. FREE SOFTWARE FOUNDATION. *GNU's not Unix – the GNU Project and the Free Software Foundation (FSF)*. `http://www.gnu.org`.
7. GARAVEL, H. Email correspondence, Jan. 2001.
8. GOTTLIEB, D., HAGERTH, S. A., LEHOT, P. G. H., AND RABINOWITZ, H. S. A classification of compression methods and their usefulness for a large data processing center. In *Proceedings of the National Computer Conference* (1975), vol. 44, pp. 453–458.
9. GROOTE, J. F., LISSER, B., AND WOUTERS, A. G. *A Language and Tool Set to Study Communicating Processes with Data*. CWI, `http://www.cwi.nl/~mcrl`, Feb. 2000.
10. HERMANNS, H., AND MERTSIOTAKIS, V. A stochastic process algebra based modelling tool. In *Performance Engineering of Computer and Telecommunications Systems* (1996), Springer Verlag.
11. KNUTH, D. E. Dynamic Huffman coding. *Journal of Algorithms 6* (1985), 163–180.
12. MADELAINE, E. Verification tools from the Concur project. *EATCS Bulletin 47* (1992).
13. MAUW, S., AND VELTINK, G. J. A tool interface language for PSF. Tech. Rep. P8912, Universiteit van Amsterdam, 1989.
14. RAMABADRAN, T. V., AND GAITONDE, S. S. A tutorial on CRC computations. *IEEE Micro* (Aug. 1988), 62–75.
15. STORER, J., AND SZYMANSKI, T. Data compression via textual substitution. *Journal of the ACM 29* (1982), 928–951.

Appendix I
SVC programming interface

data types

- SVCbool – enumerated type with constants *SVCtrue* and *SVCfalse*
- SVCfilemode – enumerated type with constants *SVCread* and *SVCwrite*
- SVCfile – structure representing an SVC file
- SVCint – integer
- SVClabelIndex – integers to denote labels
- SVCparameterIndex – integers to denote transition parameters
- SVCstateIndex – integers to denote states

external variable

- extern int SVCerrno;

file access

- int SVCopen(SVCfile **file*, char **filename*, SVCfilemode *mode*, SVCbool **indexFlag*);
  Open *filename* for access given by *mode* return in *file* the associated SVC file. For read access, in *indexFlag* is returned whether the file opened is indexed, and for write access, this parameter specifies whether the new file is indexed.
- int SVCclose(SVCfile **file*);
  Close *file* and release all resources associated.

indices

- SVClabelIndex SVCnewLabel(SVCfile **file*, ATerm *label*, SVCbool **new*);
  Registrate a action *label* for use with *file* and return in *new* whether the action is new. Return the index assigned, or the existing index if the label already existed.
- SVClabelIndex SVCaterm2Label(SVCfile **file*, ATerm *term*);
  Return the index assigned to an action label *term* or -1 if this label was not registered in *file*.

- ATerm SVClabel2ATerm(SVCfile ***file*, SVClabelIndex *index*);
  Return the aterm that has assigned *index* or NULL if no such aterm exists in *file*.
- SVCstateIndex SVCnewState(SVCfile **file*, ATerm *state*, SVCbool **new*);
- SVCstateIndex SVCaterm2State(SVCfile **file*, ATerm *term*);
- ATerm SVCstate2ATerm(SVCfile **file*, SVCstateIndex *state*);
- SVCparameterIndex SVCnewParameter(SVCfile **file*, ATerm *parameter*, SVCbool **new*);
- SVCparameterIndex SVCaterm2Parameter(SVCfile **file*, ATerm *term*);
- ATerm SVCparameter2ATerm(SVCfile **file*, SVCparameterIndex *parameter*);

transitions

- int SVCgetNextTransition(SVCfile **file*, SVCstateIndex **from*, SVClabelIndex **action*, SVCstateIndex **to*, SVCparameterIndex **parameter*);
  Read from *file* the next transition and return its components in *from*, *action*, *to* and *parameter*. If the read is successful 0 is returned, otherwise -1 is returned.
- int SVCputTransition(SVCfile **file*, SVCstateIndex *from*, SVClabelIndex *action*, SVCstateIndex *to*, SVCparameterIndex *parameter*);
  Write to *file* the transition with components *from*, *action*, *to* and *parameter*. If the write is successful 0 is returned, otherwise -1 is returned.

header

- char *SVCgetComments(SVCfile **file*);
  Return the comments from the header of *file*.
- int SVCsetComments(SVCfile **file*, char **comments*);
  Write *comments* into the header of *file*.
- char *SVCgetCreator(SVCfile **file*);
- int SVCsetCreator(SVCfile **file*, char **creator*);
- char *SVCgetType(SVCfile **file*);
- int SVCsetType(SVCfile **file*, char **type*);
- char *SVCgetVersion(SVCfile **version*);
- int SVCsetVersion(SVCfile **file*, char **version*);
- char *SVCgetFilename(SVCfile **file*);
- char *SVCgetDate(SVCfile **file*);
- char *SVCgetFormatVersion(SVCfile **file*);
- SVCint *SVCgetNumLabels(SVCfile **file*);
- SVCint *SVCgetNumStates(SVCfile **file*);
- SVCint *SVCgetNumParameters(SVCfile **file*);
- SVCint *SVCgetNumTransitions(SVCfile **file*);

error handling

- char *SVCerror(int *svcerrno*);
  Return the error message associated with error number *svcerrno*.