FAST: Fast Acceleration of Symbolic Transition systems

Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Laure Petrucci

LSV, CNRS UMR 8643
ENS de Cachan
61 avenue du président Wilson
F-94235 CACHAN Cedex
FRANCE
{bardin,finkel,leroux,petrucci}@lsv.ens-cachan.fr

Abstract. FAST is a tool for the analysis of infinite systems. This paper describes the underlying theory, the architecture choices that have been made in the tool design. The user must provide a model to analyse, the property to check and a computation policy. Several such policies are proposed as a standard in the package, others can be added by the user. FAST capabilities are compared with those of other tools. A range of case studies from the literature has been investigated.

1 Introduction

Model-checking is a widely-spread technique in critical systems verification. Several efficient model-checkers, such as SMV [SMV], SPIN [SPI] or DESIGN/CPN [CPN], are available. However, these tools are restricted to finite systems whereas many real systems are infinite, because of parameters or unbounded data structures.

FAST is a tool designed to allow automatic verification of systems modeled by automata augmented with (unbounded) integer variables (extended counter automata). The main issue addressed by FAST is the computation of the \textit{exact} (infinite) set of configurations reachable from a given set of initial configurations. Let us recall that verification of safety properties can be reduced to reachability of a given configuration from a set of initial configurations.

A lot of properties are in general undecidable, but there are two ways to deal with undecidability. The first one is to consider decidable subclasses, thus reducing the expressiveness of the model, while the second one is to accept only a semi-algorithm, which does not terminate in the general case but which is expected to terminate in most practical cases. We follow the second approach. The techniques used in FAST are based on \textit{acceleration} [FL02]. It comes down to computing the (exact) effect of iterating a control loop of an arbitrary length (cycle). These cycles are automatically chosen. Both forward and backward reachability are allowed. FAST works on \textit{linear systems}, i.e. finite sets of linear functions whose definition domains are defined by a \textit{Presburger formula over non-negative integers}. Most systems with integer variables can be described by such a system.
In [FL02], it is proved that for linear systems whose associated square matrices generate a finite multiplicative monoid — namely finite linear systems, acceleration of a loop terminates. It turns out that most integer variables systems appear to be finite linear systems. Even though termination is not guaranteed, in practice, FAST deals with a large amount of examples of our extended counter automata model (see section 4). We believe that Presburger arithmetic is sufficient to model these problems and that most systems are effectively computable.

2 Related tools

We present four main tools able to cope with integer variables infinite systems, two of them using acceleration or similar techniques.

<table>
<thead>
<tr>
<th>Variable type</th>
<th>Solvers</th>
<th>Actions</th>
<th>Acceleration</th>
<th>Presburger</th>
<th>Correct with convex sets</th>
<th>Correct with linear sets</th>
<th>Engine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Presburger</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>LNDD, MONA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>convex sets</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>NDD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>convex sets</td>
<td>no</td>
<td>-</td>
<td>yes</td>
<td>yes</td>
<td>RVA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( x_i \leq x_j + c )</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>PDBM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( x_i \leq c )</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>PDBM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( x_i \geq c )</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>PDBM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \lambda x_i \leq x_j + c )</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>period basis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \lambda x_i \leq x_j + c )</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>CST</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \lambda x_i \leq x_j + c )</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>convex polyhedra</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Presburger</td>
<td>no</td>
<td>-</td>
<td>yes</td>
<td>yes</td>
<td>OMEGA, BDD</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 1.** A comparison of different tools for reachability set computation

3 Architecture

Figure 2 shows a model of the system to analyze, a reachability property to check and a strategy to direct the computation. If it terminates, the tool answers
whether the property is satisfied or not. *Settings* can also be optionally set by
the user, such as the ordering of variables and stop criteria.

*Strategies* allow the user to direct "by-hand" the computation. Strategies
make it possible to describe standard model-checking features such as forward
or backward reachability as well as more advanced constructs like a sequence
of incremental submodel analysis. This has been successfully used to verify the
TTP protocol (see section 4). Concretely, the user describes strategies through a
high-level language allowing to manipulate Presburger definable sets of integers,
linear functions, booleans and providing primitives for pre*" and post*" operations.

Presburger definable sets of integers are internally represented by Labeled
Number Decision Diagrams (LNDs). This automata description for non-negative
integer arithmetic is based on works like XXX. LNDs allow to represent any
Presburger formula and provides basic operations on sets (intersection, nega-
tion, inclusion or emptiness test) as well as more advanced constructs like the
acceleration of a cycle described in [FL02]. Our implementation uses packages
from MONA [MON], providing automata operations. An extended version of
FAST for integer arithmetic has also been developed, but there was a drop in
performances. Since all the case studies considered only deal with non-negative
integers, we decided to first limit FAST to non-negative integers.

4 Results

FAST has been applied to a large number of examples (about 40), ranging from
Petri nets to abstract multi-threaded JAVA programs, mainly taken from [Del].
About 80% of these case studies could effectively be verified. It proves that
choices made during FAST design, like having only a semi-algorithm or restricting
FAST to non-negative integers, are sound for practical infinite systems verifica-
tion. Moreover, most of these examples require only a basic predefined strategy
(a forward search), thus only little input from the user.

Figure 3 presents the performances obtained by FAST on ten of the most
representative examples. Dekker ME is a bounded Petri net, other examples
are infinite state systems because of parameters (lift controller) or unbounded
integer variables (FMS). Despite its number of variables and transitions, the
Swimming Pool protocol is a highly non-trivial protocol. The TTP protocol is a
complex group membership protocol, using elaborate guards. The tool computes efficiently these examples. A forward search has been used for all examples. For the particular case of TTP, a more elaborate strategy was also tested, leading to considerable increase in computation time.

<table>
<thead>
<tr>
<th>Case Study</th>
<th>weakb</th>
<th>transitions</th>
<th>time(s)</th>
<th>mem(MB)</th>
<th>n. states</th>
<th>n. SCC</th>
<th>n. Reach</th>
<th>n. cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dekker ME</td>
<td>22</td>
<td>22</td>
<td>21.72</td>
<td>5.48</td>
<td>5</td>
<td>36</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>CSM</td>
<td>13</td>
<td>13</td>
<td>45.57</td>
<td>6.31</td>
<td>6</td>
<td>32</td>
<td>2</td>
<td>35</td>
</tr>
<tr>
<td>FMS</td>
<td>22</td>
<td>20</td>
<td>157.48</td>
<td>8.02</td>
<td>21</td>
<td>23</td>
<td>2</td>
<td>46</td>
</tr>
<tr>
<td>Swimming Pool</td>
<td>9</td>
<td>6</td>
<td>111</td>
<td>29.06</td>
<td>30</td>
<td>9</td>
<td>4</td>
<td>47</td>
</tr>
<tr>
<td>Producer/Consumer with Java threads - N</td>
<td>18</td>
<td>14</td>
<td>723.27</td>
<td>12.46</td>
<td>58</td>
<td>86</td>
<td>2</td>
<td>75</td>
</tr>
<tr>
<td>Lift Controller - N</td>
<td>4</td>
<td>5</td>
<td>456</td>
<td>2.90</td>
<td>14</td>
<td>4</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>TTP</td>
<td>10</td>
<td>17</td>
<td>1186.24</td>
<td>73.24</td>
<td>1140</td>
<td>16</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>TTP (ad hoc strategy)</td>
<td>10</td>
<td>17</td>
<td>246.67</td>
<td>72.87</td>
<td>1140</td>
<td>16</td>
<td>1</td>
<td>17</td>
</tr>
</tbody>
</table>

**Fig. 3.** Results using an Intel Pentium 933 MHz with 512 Mbytes

Considering the case studies that could not be verified (9 out of 40), we propose three reasons for FAST not to terminate. First of all, the input model can be such that FAST cannot terminate, either some loops have infinite associated monoids or the reachability set is not flutable, i.e. not computable using a finite set of accelerations [FL02]. Second, the computation may lead to large automata and saturate the memory. Finally, there may be too many cycles to consider, and then the heuristic used by FAST to find cycles to be accelerated reaches its limits.

**References**


[CPN] Design/CPN online. [http://www.cpsl.sdu.dk/designCPN](http://www.cpsl.sdu.dk/designCPN).


