A Machine-Checked Proof of A State-Space Construction Algorithm

Nestor Catano
The University of Madeira, M-ITI
Campus da Penteada, Funchal, Portugal
ncatano@uma.pt

Radu I. Siminiceanu
National Institute of Aerospace
100 Exploration Way, Hampton, VA 23666, USA
radu@nianet.org

Abstract

This paper presents the correctness proof of Saturation, an algorithm for generating state spaces of concurrent systems, implemented in the SMART tool. Unlike the Breadth First Search exploration algorithm, which is easy to understand and formalise, Saturation is a complex algorithm, employing a mutually-recursive pair of procedures that compute a series of non-trivial, nested local fixed points, corresponding to a chaotic fixed point strategy. A pencil-and-paper proof of Saturation exists, but a machine checked proof had never been attempted. The key element of the proof is the characterisation theorem of saturated nodes in decision diagrams, stating that a saturated node represents a set of states encoding a local fixed-point with respect to firing all events affecting only the node's level and levels below. For our purpose, we have employed the Prototype Verification System (PVS) for formalising the Saturation algorithm, its data structures, and for conducting the proofs.

1 Introduction

Software systems have become a key part of our lives, controlling or influencing many of the activities that we undertake every day. Software correctness is particularly important for safety-critical systems where people's lives can be at risk. Such systems have to be rigorously checked for correctness before they can be deployed. Several formal techniques and tools for reinforcing the dependability of critical systems exist. Temporal logic model checking is a technique to verify systems algorithmically, using a decision procedure that checks if a (usually finite) abstract model of the system, expressed as a state-transition system, satisfies a formal specification written as a temporal logic formula [7, 13]. Additionally, theorem proving is a technique to establish mathematical theorems and theories with the aid of a computer program, *i.e.*, a theorem prover. Theorem provers usually require the interaction with an experienced user to find a proof. While model checking has been successfully used to find errors, it is seldom used to seek the absolute correctness of a system or application. On the other hand, provers and proof checkers are oftentimes the most reliable means available for establishing a higher level of correctness.

For both approaches however, from the point of view of validation, and even certification, there is the outstanding issue of trustworthiness: if a formal analysis tool is employed, how does one establish the semantic validity of the analysis tool itself – in other words, *who is checking the checker?* While the formal methods community has been functioning on the premise of accumulated trust – some theorem provers and model checkers accumulate a track record of being trustworthy – the issue of certifying verification tools is of undeniable concern, the more so as experience shows that automated verification tools are far from being free of bugs [14].

Regulatory authorities (FAA, CAA, or DOD) require software development standards, such as MIL-STD-2167 for military systems and RTCA DO-178B for civil aircraft. While certified tools for generating code exist, they have been mostly confined to a very narrow segment, such as synchronous languages for embedded systems and are usually of proprietary nature.

In this paper we attempt to establish the correctness of a general purpose model checking algorithm, with the help of a theorem prover. More precisely, we present the PVS proof of correctness for Saturation [5], a non-trivial algorithm for model checking concurrent systems, implemented within the SMART [6] formal analysis tool. We have employed the Prototype Verification System (PVS) [12] for formalising the Saturation algorithm, the Multi-valued Decision Diagrams (MDDs)

[15] data structure, and for conducting the proofs. Unlike the Breadth First Search exploration algorithm which is easy to understand and formalise, Saturation is a high-performance algorithm, employing a mutually-preemptive, doubly-recursive pair of procedures that compute a series of nested local fixed points, corresponding to a *chaotic* global fixed point strategy. The key result of the proof is the characterisation theorem of saturated nodes in decision diagrams, stating that a saturated node represents a set of states encoding a local fixed-point with respect to *firing* all events affecting only the node's level and levels below. Saturation requires a *Kronecker consistent* partition of the system model in sub-models. The PVS proofs presented here take advantage of the Kronecker consistency property to express how the state-spaces generated by local next-state functions relate to the global state-space. In this regard, Saturation's correctness proof outlines an approach that may be re-used in the correctness proofs of an entire class of algorithms that rely on structural properties. Additionally, the PVS formalisation makes the invariant relating Saturation and the firing routine explicit. This is an important requirement for the SMART code. Its implementation, and the implementation of the structures it uses for storing state-spaces, must satisfy this invariant.

2 Preliminaries

Saturation employs Multi-valued Decision Diagrams (MDDs) [15], an extension of the classical Binary Decision Diagrams (BDDs), to symbolically encode and manipulate sets of states for constructing the state-space of structured asynchronous systems. MDDs encode characteristic functions for sets of the form $S_K \times \cdots \times S_1$ for systems comprising K subsystems, each with its local state space S_k, for $K \geq k \geq 1$. The particular MDD variant used in Saturation (quasi-reduced, ordered) is formally given in Definition 1.

Definition 1 (MDDs)**.** *MDDs are directed acyclic edge-labelled multi-graphs with the following properties:*

1. *Nodes are organised into $K+1$ levels from 0 to K. The expression $\langle k|p\rangle$ denotes a generic node, where k is the node's level and p is a unique index for a node at that level.*
2. *Level 0 consists of two terminal nodes $\langle 0|0\rangle$ and $\langle 0|1\rangle$, which we often denote **0** and **1**.*
3. *Level K contains only a single non-terminal node $\langle K,r\rangle$, the root, whereas levels $K-1$ through 1 contain one or more non-terminal nodes.*
4. *A non-terminal node $\langle k|p\rangle$ has $|S_k|$ arcs pointing to nodes at level $k-1$. An arc from the position $i_k \in S_k$ to the node $\langle (k-1)|q\rangle$ is denoted by $\langle k|p\rangle[i_k] = q$.*
5. *No two nodes are* duplicate, i.e.*, there are no nodes $\langle k|p\rangle$ and $\langle k|q\rangle$ such that $p{\neq}q$ and for all $i_k \in S_k$, $\langle k|p\rangle[i_k] = \langle k|q\rangle[i_k]$.*

In contrast to [15], not fully- but quasi-reduced ordered MDDs are considered in Saturation, hence *redundant* nodes, *i.e.*, nodes $\langle k|p\rangle$ such that $\langle k|p\rangle[i_k] = \langle k|p\rangle[j_k]$ for all $i_k{\neq}j_k$, are valid according to definitions used in Saturation. This relaxed requirement can potentially lead to a much larger number of nodes, but in practice this rarely occurs. On the other hand, the quasi-reduced form has the property that all the children of a node are at the same level, which can be exploited in the algorithm. Equally important, the quasi-reduced form is still canonical, as the fully-reduced form is. A sequence σ of *local states* $(i_k,\ldots,i_1)$ is referred to as a *sub-state*. Given a node $\langle k|p\rangle$, the node reached from it through a sequence σ of local states $(i_k,\ldots,i_1)$ is defined as:

$$\langle k|p\rangle[\sigma] = \begin{cases} \langle k|p\rangle & \text{if } \sigma = (), \text{the empty sequence} \\ \langle (k{-}1)|\langle k|p\rangle[i_k]\rangle[\sigma'] & \text{if } \sigma = (i_k,\sigma'), \text{with } i_k \in S_k\,. \end{cases}$$

A sub-state σ constitutes a *path* starting at node $\langle k|p\rangle$, if $\langle k|p\rangle[\sigma] = \langle 0|1\rangle$. $\mathscr{B}(k,p)$ is the set of states encoded by $\langle k|p\rangle$, *i.e.*, those states constituting paths starting at node $\langle k|p\rangle$.

$$\mathscr{B}(k,p) = \{\sigma \in S_k \times \cdots \times S_1 : \langle k|p\rangle[\sigma] = \langle 0|1\rangle\}$$

2.1 Kronecker Consistency

Saturation requires a *Kronecker consistent* partition of the system model into K sub-models. A next-state function $\mathcal{N}$ of a Kronecker structured asynchronous system model is defined on the potential state-space $\widehat{S}: S_K \times \cdots \times S_1$ decomposed by event, *i.e.*, $\mathcal{N} = \bigcup_{e\in\mathcal{E}} \mathcal{N}_e$. An event e has a *Kronecker representation* for a given model partition if $\mathcal{N}_e$ can be written as the cross-product of K local next-state functions, *i.e.*, $\mathcal{N}_e = \mathcal{N}_e^K \times \cdots \times \mathcal{N}_e^1$, where $\mathcal{N}_e^k : S_k \rightarrow 2^{|S_k|}$, for $K \geq k \geq 1$. Since S_k are all assumed to be finite in Saturation, $\widehat{S}: S_K \times \cdots \times S_1$ can be regarded as a structure of the form $\widehat{S}: \{1,\cdots,n_K\} \times \cdots \times \{1,\cdots,n_1\}$, where $n_k = |S_k|$. A partition of a system model into sub-models is *Kronecker consistent* if every event e has a Kronecker representation for that partition. The definition of $\mathcal{N}_e$ can be extended to a set of sub-states X, $\mathcal{N}_e(X) = \bigcup_{x\in X} \mathcal{N}_e(x)$, and to a set of events $\mathcal{E}$, $\mathcal{N}_{\mathcal{E}}(X) = \bigcup_{e\in\mathcal{E}} \mathcal{N}_e(X)$. An event *depends* on a particular level if the event affects the (local) state-space generated for that partition at that level. Let $Top(e)$ and $Bottom(e)$ be the highest and lowest levels an event e depends, two cases in the definition of $\mathcal{N}_{\mathcal{E}}$ result when $\mathcal{E} = \{e: Top(e) \leq k\}$ and $\mathcal{E}^k = \{e: Top(e) = k\}$. We write $\mathcal{N}_{\leq k}$ as a shorthand for $\mathcal{N}_{\{e: Top(e)\leq k\}}$, and $\mathcal{N}_{=k}$ as a shorthand for $\mathcal{N}_{\mathcal{E}^k}$. The *reachable state-space* $S \subseteq \widehat{S}$ from an initial system state X is the smallest set containing X and being closed with respect to $\mathcal{N}$, *i.e.*, $S = X \cup \mathcal{N}(X) \cup \mathcal{N}(\mathcal{N}(X)) \cdots = \mathcal{N}^*(X)$, where $*$ denotes the reflexive and transitive closure.

2.2 Saturation

Saturation relies on a routine $Fire(e,k,p)$, for exhaustively firing all the events e such that $Top(e) = k$, and recursively calling Saturation of any descendant $\langle k|p\rangle[i_k]$ of $\langle k|p\rangle$. $Fire$, like Saturation, checks whether e is enabled in a node $\langle k|p\rangle$ and then reveals and adds globally reachable states to the MDD representation of the state-space under construction. However, unlike Saturation, $Fire$ operates on a fresh node instead of modifying $\langle k|p\rangle$ in place, since $\langle k|p\rangle$ is already saturated. In Saturation, MDDs are modified locally and only between the levels on which the fired event depends on. The enabling and the outcome of firing an event e only depend on the states of sub-models $Top(e)$ through $Bottom(e)$. Definition 2 formalises the idea of a saturated node. Saturation and the routine for firing events are mutually recursive, related through the following invariant conditions: $Saturate$ is called on a node $\langle k|p\rangle$ whose children are already saturated, $Fire$ is always invoked on a saturated node $\langle l|q\rangle$ with $l < Top(e)$ and $Saturate$ is invoked just before returning from $Fire$.

Definition 2 (Saturated Node)**.** *An MDD node $\langle k|p\rangle$ is saturated if it encodes a set of states that is a fixed-point with respect to the firing of any event affecting only the node's level or lower levels, that is, if $\mathcal{B}(k,p) = \mathcal{N}^*_{\leq k}(\mathcal{B}(k,p))$.*

2.3 Saturation's Correctness

For ease of reference, Figure 1 shows the pseudo-code for the Saturation and event firing algorithms. We reproduce here the original pencil-and-paper correctness proof of the theorem relating both algorithms, for ease of reference, then proceed with the formal PVS proof.

Correctness. Let $\langle k|p\rangle$ be a node with $K \geq k \geq 1$ and saturated children, and $\langle l|q\rangle$ be one of its children with $q \neq \mathbf{0}$ and $l = k-1$; let $\mathcal{U}$ stand for $\mathcal{B}(l,q)$ before the call $Fire(e,l,q)$, for some event e with $l < Top(e)$, and let $\mathcal{V}$ represent $\mathcal{B}(l,f)$, where f is the value returned by this call; Then, $\mathcal{V} = \mathcal{N}^*_{\leq l}(\mathcal{N}_e(\mathcal{U}))$.

Proof. By induction on k. For the induction base, $k = 1$, the only possible call $Fire(e,l,\mathbf{1})$ returns $\mathbf{1}$ because of the test on l, which has value 0, in Line 1 in Figure 1. Then, $\mathcal{U} = \mathcal{V} = \{()\}$ and $\{()\} = \mathcal{N}^*_{\leq 0}(\mathcal{N}_e(\{()\}))$.
For the induction step we assume that the call to $Fire(e,l-1,\cdot)$ works correctly. Recall that $l = k-1$. $Fire$ does not add further local states to $\mathcal{L}$, since it modifies "in–place" the new node $\langle l|s\rangle$, and not node $\langle l|q\rangle$ describing the states from where the firing is explored. The call $Fire(e,l,q)$ can be resolved in three ways. If $l < Bottom(e)$, then the returned value is $f = q$ and $\mathcal{N}_e^l(\mathcal{U}) = \mathcal{U}$ for any set $\mathcal{U}$;

since q is saturated, $\mathscr{B}(l,q) = \mathscr{N}^*_{\leq l}(\mathscr{B}(l,q)) = \mathscr{N}^*_{\leq l}(\mathscr{N}_e(\mathscr{B}(l,q)))$. If $l \geq Bottom(e)$ but *Fire* has been called previously with the same parameters, then the call $Find(FCache[l], \{q,e\}, s)$ is successful. Since node q is saturated, it has not been modified further; Finally, we need to consider the case where the call $Fire(e,l,q)$ performs "real work." First, a new node $\langle l|s\rangle$ is created, having all its arcs initialised to $\mathbf{0}$. We explore the firing of e in each state i satisfying $\langle l|q\rangle[i] \neq \mathbf{0}$ and $\mathscr{N}_l{}^e(i) \neq \emptyset$. By induction hypothesis, the recursive call $Fire(e, l-1, \langle l|q\rangle[i])$ returns $\mathscr{N}^*_{\leq l-1}(\mathscr{N}_e(\mathscr{B}(l-1, \langle l|q\rangle[i])))$. Hence, when the "while $\mathscr{L} \neq \emptyset$" loop terminates, $\mathscr{B}(l,s) = \bigcup_{i \in \mathscr{S}^l} \mathscr{N}^l_e(i) \times \mathscr{N}^*_{\leq l-1}(\mathscr{N}_e(\mathscr{B}(l-1, \langle l|q\rangle[i]))) = \mathscr{N}^*_{\leq l-1}(\mathscr{N}_e(\mathscr{B}(l,q)))$ holds. Thus, all children of node $\langle l|s\rangle$ are saturated. According to the induction hypothesis, the call $Saturate(l,s)$ correctly saturates $\langle l|s\rangle$.

Therefore, we have $\mathscr{B}(l,s) = \mathscr{N}^*_{\leq l}(\mathscr{N}^*_{\leq l-1}(\mathscr{N}_e(\mathscr{B}(l,q))) = \mathscr{N}^*_{\leq l}(\mathscr{N}_e(\mathscr{B}(l,q)))$ after the call. □

3 The PVS Formalisation

We used the Prototype Verification System (PVS) [12] for formalising *Saturate*, the MDD data structure it uses to store state-spaces, and for conducting the correctness proofs. Our formalisation is purely functional, e.g., we do not formalise memory and memory operations. The main goal of our PVS formalisation is to machine-check the pencil-and-paper correctness proof of *Saturate*, introduced in Section 2.3 and Figure 1. Modelling memory is necessary when one is interested in generating actual code from the formalisation. We are planning to port our PVS formalisation to AtelierB [2] and use refinement calculus techniques to generate Saturation actual code from the B model. Yet, this is future work.

We started formalising the basic concepts employed by the definition of Kronecker consistency such as events, states, local state values, next-state functions, local next-state functions. Then, we formalised the Saturation algorithm and the routine for firing events, and conducted their correctness proof in PVS following the pencil-and-paper proof. In the following, we present the definition of some of those basic concepts in PVS. Predicate `local_value?(m)` below formalises local state values at level `m`, where `nk` is the function $n_k = |S_k|$. The reader should remember that *Saturate* generates state spaces of values on a level basis. This is a direct consequence of the definition of Kronecker consistency. `local_value(m)` is the type of all elements satisfying the predicate `local_value?(m)`, and the type `state(k)` formalises sub-states of size `k` as sequences `s` whose elements `s`sq(m)` at any position `m` $\leq$ `k` are restricted by `nk`. We further define `s_t(k,s,m)` (not shown here) to be a sub-state of `s` of size `m` $\leq$ `k`, where `k` is the size of `s`.

```
local_value?(m)(n): bool = (m=0 ∧ n=0) ∨ (m > 0 ∧ n > 0 ∧ n ≤ nk(m))
local_value(m): type = (local_value?(m))
state(k): type = { s:Seq(k) | ∀(m:upto(k)): (m=0 ∧ s`sq(m)=0) ∨
                                            (m > 0 ∧ s`sq(m) > 0 ∧ s`sq(m) ≤ nk(m)) }
```

We use PVS datatypes to model MDDs, formed using *type constructors*, and define predicates `ordered?` and `reduced?` (not shown here), modelling ordered and reduced MDDs. Predicate `reduced?` subsumes `ordered?`. The type `OMDD` below formalises Definition 1. We further define function `level`, which returns the height of an MDD node, `child(p,i)`, which returns the `i`-th child of an MDD `p`, and predicate `trivial?` that holds of an MDD if it is **0** or **1**.

```
OMDD: type = (reduced?)
```

Events are formalised as the type `event` below with two functions `Top` and `Bottom` returning the highest and lowest level an event depends upon. The symbol `+` in the type definition indicates that the type `event` is non-empty. `TopLesser(k)` models the set $\mathscr{E} = \{e\colon Top(e) \leq k\}$.

```
event: type+

Top:[event -> posnat]
Bottom:[event -> posnat]

TopLesser(k): setof[event] = {e:event | Top(e) ≤ k}
```

```
$Saturate$(in $k$:$level$, $p$:$index$)
  Update $\langle k|p\rangle$ in–place, to encode $\mathcal{N}^*_{\leq k}(\mathcal{B}(\langle k|p\rangle))$.

  declare $e$:$event$;
  declare $\mathcal{L}$:set of $local$;
  declare $f,u$:$index$;
  declare $i,j$:$local$;
  declare $pChanged$:$bool$;

   1. repeat
   2.   $pChanged \leftarrow false$;
   3.   foreach $e \in \mathcal{E}^k$ do
   4.     $\mathcal{L} \leftarrow Locals(e,k,p)$;
   5.     while $\mathcal{L} \neq \emptyset$ do
   6.       $i \leftarrow Pick(\mathcal{L})$;
   7.       $f \leftarrow Fire(e,k-1,\langle k|p\rangle[i])$;
   8.       if $f \neq \mathbf{0}$ then
   9.         foreach $j \in \mathcal{N}^k_e(i)$ do
  10.           $u \leftarrow Union(k-1,f,\langle k|p\rangle[j])$;
  11.           if $u \neq \langle k|p\rangle[j]$ then
  12.             $\langle k|p\rangle[j] \leftarrow u$;
  13.             $pChanged \leftarrow true$;
  14.             if $\mathcal{N}^k_e(j) \neq \emptyset$ then
  15.               $\mathcal{L} \leftarrow \mathcal{L} \cup \{j\}$;
  16. until $pChanged = false$;
```

```
$Union$(in $k$:$level$, $p$:$index$, $q$:$index$):$index$
  Build an MDD rooted at $\langle k|s\rangle$ encoding $\mathcal{B}(\langle k|p\rangle) \cup \mathcal{B}(\langle k|q\rangle)$. Return $s$.

  declare $i$:$local$;
  declare $s,u$:$index$;

   1. if $p = \mathbf{1}$ or $q = \mathbf{1}$ then return $\mathbf{1}$;
   2. if $p = \mathbf{0}$ or $p = q$ then return $q$;
   3. if $q = \mathbf{0}$ then return $p$;
   4. if $Find(UCache[k],\{p,q\},s)$ then
      return $s$;
   5. $s \leftarrow NewNode(k)$;
   6. for $i = 0$ to $n^k-1$ do
   7.   $u \leftarrow Union(k-1,\langle k|p\rangle[i],\langle k|q\rangle[i])$;
   8.   $\langle k|s\rangle[i] \leftarrow u$;
   9. $Check(k,s)$;
  10. $Insert(UCache[k],\{p,q\},s)$;
  11. return $s$;
```

```
$Fire$(in $e$:$event$, $l$:$level$, $q$:$index$):$index$
  Build an MDD rooted at $\langle l|s\rangle$ encoding $\mathcal{N}^*_{\leq l}(\mathcal{N}_e(\mathcal{B}(\langle l|q\rangle)))$. Return $s$.

  declare $\mathcal{L}$:set of $local$;
  declare $f,u,s$:$index$;
  declare $i,j$:$local$;
  declare $sChanged$:$bool$;

   1. if $l < Bottom(e)$ then return $q$;
   2. if $Find(FCache[l],\{q,e\},s)$ then
   3.   return $s$;
   4. $s \leftarrow NewNode(l)$;
   5. $sChanged \leftarrow false$;
   6. $\mathcal{L} \leftarrow Locals(e,l,q)$;
   7. while $\mathcal{L} \neq \emptyset$ do
   8.   $i \leftarrow Pick(\mathcal{L})$;
   9.   $f \leftarrow Fire(e,l-1,\langle l|q\rangle[i])$;
  10.   if $f \neq \mathbf{0}$ then
  11.     foreach $j \in \mathcal{N}^l_e(i)$ do
  12.       $u \leftarrow Union(l-1,f,\langle l|s\rangle[j])$;
  13.       if $u \neq \langle l|s\rangle[j]$ then
  14.         $\langle l|s\rangle[j] \leftarrow u$;
  15.         $sChanged \leftarrow true$;
  16. if $sChanged$ then
  17.   $Saturate(l,s)$;
  18. $Check(l,s)$;
  19. $Insert(FCache[l],\{q,e\},s)$;
  20. return $s$;
```

```
$Locals$(in $e$:$event$, $k$:$level$, $p$:$index$):
set of $local$
  Return $\{i \in \mathcal{S}^k : \langle k|p\rangle[i] \neq \mathbf{0}, \mathcal{N}^k_e(i) \neq \emptyset\}$,
  the local states in $p$ locally enabling $e$.
  Return $\emptyset$ or $\{i \in \mathcal{S}^k : \mathcal{N}^k_e(i) \neq \emptyset\}$, respectively, if $p$ is $\mathbf{0}$ or $\mathbf{1}$.
```

```
$Check$(in $k$:$level$, inout $p$:$index$)
  If $\langle k|p\rangle$ is the duplicate of an existing $\langle k|q\rangle$ delete $\langle k|p\rangle$ and set $p$ to $q$.
  Else, insert $\langle k|p\rangle$ in the unique table.
  If $\langle k|p\rangle[0] = \cdots = \langle k|p\rangle[n^k-1] = \mathbf{0}$ or $\mathbf{1}$,
  delete $\langle k|p\rangle$ and set $p$ to $\mathbf{0}$ or $\mathbf{1}$, since
  $\mathcal{B}(\langle k|p\rangle)$ is $\emptyset$ or $\mathcal{S}^k \times \cdots \times \mathcal{S}^1$, respectively.
```

Figure 1: Pseudo–code for the node–saturation and event firing algorithms.

The type `next(k)` describes $\mathcal{N}_e$ for some event e. Local next-state functions at level `k` are introduced by the type `Localnext(k)`. A finite sequence of local next-state functions `fs` makes a next-state function `N` Kronecker consistent, `Kronecker?(k)(N)(fs)`, if for each event `e`, and sub-states `x` and `y` of size `k`, every `y`'s local state `y`sq(m)` with $\mathtt{m} \leq \mathtt{k}$ is an image of `x`'s local state `x`sq(m)` through the local next-state function `fs`sq(m)(e)` (modelling $\mathcal{N}_e^m$).

```
next(k): type = [ event ->  [state(k) -> setof[state(k)]] ]
Localnext(k): type = [ event  -> [local_value(k)  -> setof[local_value(k)]] ]

Kronecker?(k)(N)(fs): bool =
 ∀(e:event, x,y:state(k)):
   N(e)(x)(y) ⇔ ∀(m:upto(k)): fs`sq(m)(e)(x`sq(m))(y`sq(m))
```

`NextLesser(k)(N,fs)(x)` formalises $\mathcal{N}_{\leq k}$ applied over a sub-state x consisting of `k` levels. The formalisation assumes that `fs` is a sequence of local next-state functions that make `N` Kronecker consistency. That is, $\mathcal{N}_e$ has a Kronecker representation for all event e such that $Top(e) \leq k$. We further define `NextLesser(k,m)(N,fs)` similar to `NextLesser(m)(N,fs)` (not shown here) for levels less than or equal to `m`, and equals to the identity function from `m+1` to `k`. Definition of `NextLesser(m)(N,fs)` is extended to a set `X` of sub-states in the natural way.

```
NextLesser(k)(N, fs)(x): setof[state(k)] =
 { y:state(k) | ∃(e:(TopLesser(k))):
                 ∀(m:upto(k)): fs`sq(m)(e)(x`sq(m))(y`sq(m)) }
```

$\mathcal{N}^*_{\leq k}(X)$ is formalised as `Apply(k)(N,fs)(w)(X)` below, where `w` represents the number of iterations after which applying $\mathcal{N}_{\leq k}$ on X does not generate any new state. That is, X is a fixed-point of $\mathcal{N}_{\leq k}$. The existence of such `w` is guaranteed by our formalisation since X is a finite set, and every $\mathcal{N}_e$ in $\mathcal{N}_{\leq k}$ with $Top(e) \leq k$ is an increasing function. A similar definition for `Apply(k,m)(N,fs)(w)(X)` exists (not shown here) that uses `NextLesser(k,m)(N,fs)(X)` instead of `NextLesser(k)(N,fs)(X)`. As a consequence of the definition of $\mathcal{N}^*_{\leq k}$, and because we are considering Kronecker consistency next-state functions, we have that $\mathcal{N}^*_{\leq k}(\mathcal{N}^*_{\leq k-1}(X)) = \mathcal{N}^*_{\leq k}(X)$ (Corollary `Apply_cor1`).

```
Apply(k)(N,fs)(w)(X): recursive setof[state(k)] =
 if w=0 then X elsif w=1 then union(X,NextLesser(k)(N,fs)(X))
 else union(X,Apply(k)(N,fs)(w-1)(X)) endif
measure w

Apply_cor1: corollary
 ∀(k:{n:upto(K) | n > 0}, N:next(k), fs:(kronecker?(k)(N)),
   w:posnat, X:setof[state(k)], s:state(k)):
 Apply(k)(N,fs)(w)(Apply(k,k-1)(N,fs)(w)(X))(s) ⇔ Apply(k)(N,fs)(w)(X)(s)
```

Finally, we formalise the routine for firing events and model its usage in Saturation. We employed an axiomatic approach rather than writing a definition for the routine. Although axioms might potentially introduce inconsistencies, the use of definitions may force PVS to generate additional proof obligations all over the lemmas and theorems using the definitions, cluttering their proofs. Furthermore, an axiomatic approach is preferable when one is not interested in generating code for the algorithm directly. The firing of an event e on a node $\langle l|q\rangle$, `fire(l,e,N,fs,w,q)` is described by axioms `fire_trivial`, `fire_nontrivial`, `fire_recursive`, and `fire_ saturated` below, with $\mathtt{l} \leq \mathtt{K}$, `e:{ev:event | l < Top(ev)}`, `N:next(l)`, `fs:(Kronecker?(l )(N))`, and `q:{u:OMDD | level(u)=l` $\wedge$ `saturated?(l,u)(N,fs)(w)}`. Therefore, to be able to call `fire(l,e,N,fs,w,q)`, event `e` should be such that `l < Top(e)`, and `q` should be such that `saturated?(l,q)(N, fs)(w)`, in accordance with the "*Fire* is always invoked on a saturated node $\langle l|q\rangle$ with $l < Top(e)$" invariant condition. `fire_trivial` states that `fire(l,e,N,fs,w,q)` returns `q` unaffected when either `l < Bottom(e)` or `l = 0`. These two conditions describe the cases when the recursive call to `fire(l,e,N,fs,w,q)` ends. `fire_nontrivial` states that if `l > 0`, then `fire(l,e,N,fs,w,q)`'s call returns a node whose level is the same as q's level, that is l. trivial? holds of an MDD if it is **0** or **1**. fire_recursive states that the set Below(l,fire(l,e,N,fs,w,q)) of states of size l encoded by a call to fire(l,e,N,fs,w,q) is recursively generated on a Kronecker structured level-basis. That is, at level l, the local next-state function $\mathcal{N}_e^l$ (see fs`sq(l)(e)) generates all the local states for every local state value i at level l, and recursively, fire(l-1,e,$N_{l'}$,$fs_{l'}$,w,child(q,i)) generates all the sub-states of size l-1. The final challenge in our formalisation comes from modelling the mutual recursion between *Saturate* and *Fire*. Mutual recursion is not directly supported by PVS. The axiom fire_saturated formalises the invariant "*Saturate* is invoked just before returning from *Fire*" in which we model Saturated as a property of a node that is fired on a particular event. Notice that fire_saturated cannot directly be expressed as a definition.

```
fire(l,e,N,fs,w,q): OMDD

fire_trivial: axiom l < Bottom(e) ∨ l=0 ⇒ fire(l,e,N,fs,w,q)=q

fire_nontrivial: axiom
 l > 0 ⇒ ¬trivial?(fire(l,e,N,fs,w,q)) ∧ level(fire(l,e,N,fs,w,q))=l

fire_recursive: axiom
 Below(l,fire(l,e,N,fs,w,q)) =
  { s:state(l) | ∃(i:local_value(l)): fs`sq(l)(e)(i)(s`sq(l)) ∧
             Below(l-1,fire(l-1,e,N_l',fs_l',w,child(q,i)))(s_t(l,s,l-1)) }

fire_saturated: axiom saturated?(l,fire(l,e,N,fs,w,q))(N,fs)(w)
```

3.1 Saturation's Formalisation

Theorem 1 formalises $\mathcal{N}^*_{\leq k-1}(\mathcal{N}_e(\mathcal{B}(k,p))) = \bigcup_{i\in S^k} \mathcal{N}_e^k(i) \times \mathcal{N}^*_{\leq k-1}(\mathcal{N}_e(\mathcal{B}(\langle k-1|\langle k|p\rangle[i]\rangle)))$, which is true in Kronecker systems.

Theorem 1 (Applying Kronecker Consistent Next-State Functions). *This theorem is used in Saturation's correctness proof (Theorem 2).*

```
kronecker_apply: theorem
 ∀(k:{n:upto(K) | n > 0}, p:{u:OMDD | ¬trivial?(u) ∧ level(u)=k}, ev:event,
   N:next(k), fs:(kronecker?(k)(N)), w:posnat, w_1:posnat, s:state(k)):
    ( ∃(i:local_value(k)):
        fs`sq(k)(ev)(i)(s`sq(k)) ∧
        Apply(k-1)(N_k',fs_k')(w_1)(Next(k-1,ev)(N_k',fs_k')(Below(k-1,child(p,i))))
                              (s_t(k,s,k-1)) )
        ⇔
    Apply(k,k-1)(N,fs)(w)(Next(k,ev)(N,fs)(Below(k,p)))(s)
```

The proof of Theorem 1 is conducted under the well_defined assumption below, which states that since "the enabling and the outcome of firing an event e only depend on the states of sub-models $Top(e)$ through $Bottom(e)$", if $k < Bottom(e)$ then applying $\mathcal{N}_{\leq k}$ to $\mathcal{B}(k,p)$ does not generate any new state, and the non_decreasing assumption, which states that all considered local next-state functions f are non-decreasing. Below(k,p) formalises $\mathcal{B}(k,p)$, the set of states encoded by $\langle k|p\rangle$.

```
well_defined: assumption
 ∀(k:upto(K), p:{u:OMDD | level(u)=k}, e:event,
   N:next(k), fs:(Kronecker?(k)(N))):
  k < Bottom(e) ⇒ Below(k,p) = Next(k,e)(N,fs)(Below(k,p))

non_decreasing: assumption
  ∀(k:upto(K), e:event, f:Localnext(k), i:local_value(k)): f(e)(i)(i)
```

Theorem 1 is proved by induction on `w` and `w`$_1$. The base case, `w=1` and `w`$_1$`=1`, is proved from the lemma `below_incremental` below. This lemma states that a state `s` of size `k>0` belongs to `Below(k,p)` if and only if `p` is not trivial (`p` is different to **0** and **1**), and `s_t(k,s,k-1)` belongs to `Below(k-1,child(n,s`sq(s`ln)))`.

```
below_incremental: lemma
  Below(k,p)(s) ⇔
  ( ¬trivial?(p) ∧ Below(k-1,child(p,s`sq(s`ln)))(s_t(k,s,k-1)) )
```

The inductive step for Theorem 1 is shown below. It reduces after expanding the definition of `Apply` in the `w+1` part of the logical equivalence.

```
Apply(k,k-1)(N,fs)(w)(Next(k,ev)(N,fs)(Below(k,p)))(s)
  ⇔
Apply(k,k-1)(N,fs)(w+1)(Next(k,ev)(N,fs)(Below(k,p)))(s)
```

Theorem 2 below formalises Saturation's correctness condition in PVS. The PVS proof of this theorem follows the pencil-and-paper proof in Section 2.3.

Theorem 2 (Saturation's Correctness)**.** *$\mathscr{B}(l,f) = \mathscr{N}^*_{\leq l}(\mathscr{N}_e(\mathscr{U}))$, where f is the value returned by Fire(l,e,q)'s call, and $\mathscr{U}$ stands for $\mathscr{B}(l,q)$ before calling Fire(l,e,q), for some event e such that $l < Top(e)$.*

```
saturation_correctness: theorem
 ∀(l:upto(K), e:{ev:event | l < Top(ev)}, N:next(l), fs:(Kronecker?(l)(N),
   w:posnat, q:{u:OMDD | level(u)=l ∧ saturated?(l,u)(N,fs)(w)})):
  Below(l,fire(l,e,N,fs,w,q)) =
   Apply(l)(N,fs)(w)(Next(l,e)(N,fs)(Below(l,q)))
```

Proof. By induction on `l`.

(*i.*) Base case (`l=0`). If `l=0` then `fire(l,e,N,fs,w,q) = q` (Axiom `fire_trivial`). `Below(0,q)` equals `TSeq` (the empty sequence), and `Next(0,e)(N,fs)({TSeq}) = {TSeq}`. Because `Apply(0)(N,fs)(w)({TSeq}) = {TSeq}`, the base case reduces trivially.

```
level(q)=0 ∧ 0 < Top(e) ∧ reduced?(q) ∧
Kronecker?(0)(N)(fs) ∧ saturated?(0,q)(N,fs)(w)
  ⇒
Below(0,fire(0,e,N,fs,w,q)) =
 Apply(0)(N,fs)(w)(Next(0,e)(N,fs)(Below(0,q)))
```

(*ii.*) Inductive step with `f = fire(l+1,e,N,fs,w,q)`.

```
l+1 < Top(e) ∧ level(q)=l+1 ∧ reduced?(q) ∧
Kronecker?(l+1)(N)(fs) ∧ saturated?(l+1,q)(N,fs)(w) ∧
(∀(ee:{ev:event | l < Top(ev)}, N₁:next(l), fs₁:(Kronecker?(l)(N₁)),
   ww:nat, qq:{u:OMDD | level(u)=l ∧ saturated?(l,u)(N₁,fs₁)(ww)}):
  Below(l,fire(l,ee,N₁,fs₁,ww,qq)) =
    Apply(l)(N₁,fs₁)(ww)(Next(l,ee)(N₁,fs₁)(Below(l,qq))))
  ⇒
Below(l+1,f) = Apply(l+1)(N,fs)(Next(l+1,e)(N,fs)(Below(l+1,q)))
```

(*ii.i*) Let us suppose `l+1 < Bottom(e)`. From axiom `fire_trivial`, `f = fire(l+1,e,N,fs, w,q) = q`. The proof is discharged from this, `saturated?(l+1,q)(N,fs)(w)` in the antecedent of the proof, and the assumption `well_defined`.

(*ii.ii*) Let us suppose `l+1` $\geq$ `Bottom(e)`. If `trivial?(q)` then `level(q) = 0`, which contradicts the hypothesis `level(q) = l+1`. We hence assume `¬trivial?(q)` afterwards. From the axiom `fire_saturated` and `saturated?(l,f)(N,fs)(w)` in the hypothesis of the proof, `Below(l+1,f) = Apply( l+1)(N,fs)(w)(Below(l+1,f))`. Because

`Below(l+1,f) = Apply(l+1,l)(N,fs)(w)(Next(l+1,e)(N,fs)(Below(l+1,q)))`[1], then `Below(l+1,f) = Apply(l+1)(N,fs)(w)(Apply(l+1,r)(N,fs)(w)(Next(l+1, e)(N,fs)(Below(l+1,q))))`. Therefore, from Corollary `Apply_cor1` in Section 3, `Below(l+1,f) = Apply(l+1)(N,fs)(Next(l+1, e)(N,fs) (Below(l+1,q)))`.

□

4 Conclusion and Future Work

Saturation is a high-performance non-trivial algorithm with an existing pencil-and-paper correctness proof. Conducting Saturation's correctness proof in PVS allowed us to verify correct the existing pencil-and-paper proof. Additionally, the PVS type-checker ensures that all the definitions in Saturation are type-correct, and that details are not overlooked. The Kronecker consistency property of systems considered in Saturation allows a separation of concerns so that proof-constraints did not clutter the actual structural proofs we conducted. In this regard, Saturation's correctness proof outlines a proof approach for an entire family of algorithms relying on structural properties. However, there is still a missing link. We proved the correctness of a model of Saturation. But, how do we know that Saturation's implementation faithfully attests to this model? As future work, we will pursue research in generating Java or C code from the PVS formalisation of Saturation, in the spirit of C. Muñoz and L. Lensink's work in [11], and comparing this code with the existing implementation of Saturation in the SMART formal analysis tool.

The full formalisation of Saturation in PVS consists of 7 theories, 10 lemmas, 7 corollaries, 2 main theorems, and 107 Type-Correctness Conditions (TCCs). The full formalisation can be reached at `http://www.uma.pt/ncatano/satcorrectness/saturation-proofs.htm`. Our formalisation is purely functional, e.g., we do not formalise memory, or memory operations.

Future Work. In [11], Muñoz and Lensink present a prototype code generator for PVS which translates a subset of PVS functional specifications into the Why language [8] then to Java code annotated with JML specifications [3, 4]. However, the code generator is still a proof of concept so that many of its features have to be improved. We will pursue research in that direction so as to generate Java certified code from the PVS formalisation of Saturation, and compare this with the existing implementation of Saturation in the SMART formal analysis tool.

In a complementary direction, our PVS formalisation of Saturation can be ported into B [1]. Then, using refinement calculus techniques [9, 10], e.g., implemented in the AtelierB tool [2], code implementing Saturation can be generated. This code is ensured to comply with the original formalisation of Saturation. A predicate calculus definition would require that axiomatisation for the routine for firing events (Section 3) is replaced by a more definitional style of modelling.

References

[1] J. R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] AtelierB. `http://www.atelierb.eu/index_en.html`.

[3] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.

[4] N. Catano, F. Barraza, D.García, P. Ortega, and C. Rueda. A case study in JML-assisted software development. In *SBMF: Brazilian Symposium on Formal Methods*, 2008.

[5] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state-space generation. In *Tools and Algorithms for the Construction and Analysis of*

[1]This result is not proved here.

Systems (TACAS), volume 2031 of *Lecture Notes in Computer Science*, pages 328–342, Genova, Italy, April 2001. Springer-Verlag.

[6] Gianfranco Ciardo, R. L. Jones III, Andrew S. Miner, and Radu Siminiceanu. Logic and stochastic modeling with SMART. *Journal of Performance Evaluation*, 63(6):578–608, 2006.

[7] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications. In *Proceedings of TOPLAS*, pages 244–263, 1986.

[8] J.-C. Filliâtre and Claude Marché. Multi-prover verification of c programs. In *International Conference on Formal Engineering Methods (ICFEM)*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29, 2004.

[9] J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In *European Symposium on Programming (ESOP)*, pages 187–196, 1986.

[10] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

[11] Leonard Lensink, César Muñoz, and Alwyn Goodloe. From verified models to verifiable code. Technical Memorandum NASA/TM-2009-215943, NASA, Langley Research Center, Hampton VA 23681-2199, USA, June 2009.

[12] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, Jun 1992. Springer-Verlag.

[13] A. Pnueli. The temporal logic of programs. In *Symposium on the Foundations of Computer Science (FOCS)*, pages 46–57, Providence, Rhode Island, USA, 1977. IEEE Computer Society Press.

[14] K. Y. Rozier and M. Vardi. LTL satisfiability checking. In *SPIN*, pages 149–167, 2007.

[15] R. K. Brayton Timothy Y.K. Kam, T. Villa and A. L. Sangiovanni-Vincentelli. Multi-valued decision diagrams: Theory and applications. *Multiple-Valued Logic*, 4(1–2), 1998.