Publication II

Tommi Junttila and Jori Dubrovin. Encoding Queues in Satisfiability Modulo Theories Based Bounded Model Checking. In *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference (LPAR 2008)*, pages 290–304, November 2008.

Encoding Queues in Satisfiability Modulo Theories Based Bounded Model Checking*

Tommi Junttila and Jori Dubrovin

Helsinki University of Technology TKK
Department of Information and Computer Science
P.O. Box 5400, FIN-02015 TKK, Finland
Tommi.Junttila@tkk.fi, Jori.Dubrovin@tkk.fi

Abstract. Using a Satisfiability Modulo Theories (SMT) solver as the back-end in SAT-based software model checking allows common data types to be represented directly in the language of the solver. A problem is that many software systems involve first-in-first-out queues but current SMT solvers do not support the theory of queues. This paper studies how to encode queues in the context of SMT-based bounded model checking, using only widely supported theories such as linear arithmetic and uninterpreted functions. Various encodings with considerably different compactness and requirements for available theories are proposed. An experimental comparison of the relative efficiency of the encodings is given.

1 Introduction

Bounded model checking (BMC) [1] is an efficient symbolic model checking technique that has been successfully applied to finding bugs in hardware, software, timed, and hybrid systems. In a recent industrial project we have applied BMC to the analysis of asynchronous, message passing, object oriented systems described in UML [2,3]. Such systems rise naturally e.g. in the context of communication protocol design. The proposed BMC techniques seem to be relatively efficient (especially when the so-called step semantics are applied [3]) and sometimes even complementary to the explicit state methods traditionally used in the analysis of this kind of systems. In [2,3] we use the NuSMV tool [4] as the back-end, and thus the symbolic transition relation is eventually translated ("bit-blasted") into propositional logic and solved with a propositional satisfiability (SAT) solver. Compared to propositional SAT, Satisfiability Modulo Theories (SMT, see e.g. [5,6,7]) offers an attractive framework for solving problems involving constraints over non-Boolean domains such as linear arithmetics over reals or integers, equality with uninterpreted functions (EUF), lists, arrays, and so on. Encouraged by this and the tremendous improvements in the efficiency of SMT solvers during the last few years, we have also implemented and experimented with an SMT-based variant of our UML BMC encoding.

When applying BMC to asynchronous message passing systems, one has to be able to encode *queues* in symbolic form accepted by SMT solvers. Unfortunately, decision

* This work has been financially supported by the Academy of Finland (project 112016), Helsinki Graduate School in Computer Science and Engineering, and Jenny and Antti Wihuri Foundation.

I. Cervesato, H. Veith, and A. Voronkov (Eds.): LPAR 2008, LNCS 5330, pp. 290–304, 2008.

procedures for theories of queues, especially with sub-queue relations, can be quite complex (see e.g. [8]) and, to the authors' knowledge, are not currently implemented in any of the state-of-the-art SMT solvers. However, in the context of finding counter-examples to safety properties of message passing systems with BMC, we do not need a full theory of queues but *only* enqueue and dequeue operations.[1] If we wish to check liveness properties with BMC (see e.g. [11]) or use temporal induction [12,13] to prove absence of bugs as well, we also need a predicate that checks whether the contents of a queue are the same at two different time steps. As current SMT solvers do not support these restricted theories of queues either (the theory of recursive data structures [14] and its variants implemented in some SMT solvers are not applicable, as they only support stacks and Lisp-like lists, i.e. last-in-first-out protocol instead of first-in-first-out), we have developed ways to encode queues with other theories. In this paper we study how to do such symbolic queue encodings in the BMC context by using fragments of quantifier-free first order logic supported by the current state-of-the-art SMT solvers. Our goal is to develop queue encodings that (i) are compact (queue encodings can form a significant part of the symbolic transition relation encoding used in BMC), (ii) only require theories that are supported in the current SMT solvers, and (iii) are hopefully efficient to solve. We present several alternative queue encodings that vary considerably in compactness and in what kind of theories they apply; we mainly concentrate on queues with fixed bounded capacity but also present one (very compact) encoding that can handle unbounded queues. We benchmark the proposed encodings by using a simple scalable "stress test" model and some real UML models. Naturally, our queue encodings can also be applied to BMC of any hardware or software system that uses queues, not only message passing protocols.

Related work. Compared to some other theories such as those of arrays or linear arithmetics, there seems to be relatively little work on developing and implementing decision procedures for queues.

In [15], lambda functions are used to describe queues within the context of microprocessor verification. However, the expansion of the lambda functions with beta-substitution, required for getting an SMT problem without lambdas, seems to result in a quadratic blow-up with respect to the BMC bound.

As a part of his thesis [8, Chapter 8], Bjørner develops a decision procedure for queues. However, concatenation of queues as well as sub-queue relations are considered, making the decision procedure rather involved compared to our needs. To our knowledge, it is not implemented in any state-of-the-art SMT solver. Based on the axioms given in [8, Chapter 8], one possibility would be to use the cons/revcons constructors to describe queue contents, then eliminate the revcons constructors by using the axioms, and finally solve the resulting problem with an SMT-solver supporting the theory of recursive data types (e.g. Yices [16] to name just one example of such a solver). However, eliminating the revcons constructors in this way leads to a quadratic explosion in the size of the formula with respect to the BMC bound.

[1] Actually, state machines in UML [9] (as well as in SDL [10]) can also temporarily defer messages; the symbolic translation in [2] can handle this, but due to space limitations we do not consider deferring in this paper.

The queue interface concept we use in this paper is similar to the one proposed in [17,18] for encoding memories in the context of BMC for embedded systems. It seems that queues are easier than memories in this setting as the constraints for memories in [17,18] depend on the BMC bound, making the size of the overall encoding quadratic with respect to the bound; the queue encodings presented in this paper are linear with respect to the bound.

As a final note it should be noticed that some of the underlying high-level concepts in our queue encodings are by no means new. Anyone who has programmed a queue data structure with a programming language such as C or C++ has certainly considered the basic ideas of both the "shifting" and "cyclic" approaches of this paper. But we are not aware of any previous attempts to systematically describe, analyze, and benchmark these approaches in the context of SMT-based BMC. Furthermore, the "linear" approach and the "tag-based" element compression exploit uninterpreted functions for reducing the problem size in, we believe, a novel way.

2 BMC and the Queue Interface

In Model Checking [19], we can consider a system to be composed of a finite vector $s = \langle x_1, \ldots, x_n \rangle$ of typed *state variables*, the set $I \subseteq S$ of *initial states*, and the *transition relation* $R \subseteq S \times S$, where $S = domain(x_1) \times \cdots \times domain(x_n)$ is the set of *states* of the system. A pair $\langle s, s' \rangle$ is in the transition relation iff the system can move from s to s' in one execution step. We will primarily consider checking *invariant properties* of systems and define the set $B \subseteq S$ of *bad states*, in which the invariant is broken. The model checking problem is to determine whether a bad state can be reached from any initial state with a finite number of transitions.

In Bounded Model Checking (BMC), the characteristic functions of the sets I, R, and B are encoded as formulas $I(s)$, $R(s, s')$, and $B(s)$ over vectors of state variables. The question is then whether there is a *bound* $K \geq 0$ and a sequence $s_0, \ldots, s_K$ of states such that $I(s_0) \wedge R(s_0, s_1) \wedge \cdots \wedge R(s_{K-1}, s_K) \wedge B(s_K)$ holds. For a fixed bound, a satisfiability checker, in this case an SMT solver, can decide the existence of a state sequence that satisfies the latter formula and thus show whether a bad state can be reached.

The problem we address is that if one or more of the state variables represent queues, there is no direct way of encoding the queue contents and operations in the language of the currently available SMT solvers. We propose several alternative queue encodings that only use theories that are widely supported by state-of-the-art solvers. We will encapsulate each queue behind a unified interface that allows (limited) access to the contents of the queue, making it possible to switch to a different queue encoding while keeping the encoding of the rest of the system the same. Although we only talk about a single queue in a system, several queues can be handled by simply duplicating the interface and the encoding.

We assume that at each time step, each queue can be the target of at most one enqueue and at most one dequeue operation (both can occur at the same time step provided that the queue is not empty: that is, the element enqueued at a time step cannot be dequeued at the same time step but only later).

The interface consists of two client-controlled Boolean variables, which tell whether a dequeue/enqueue operation is executed, and two data elements: one that tells the client of the queue what the first element in the queue is at the current time step, and another for the element to be appended into the queue in the case the enqueue operation is executed. Furthermore, the client can query whether the queue is empty or full, and whether the contents of the queue at two different time steps match. As mentioned in the introduction, the equality test between time steps is only required if one wishes to check liveness properties or apply temporal induction (see e.g. [11,12,13]).

The contents of a queue with element type ELEM can be represented as a variable-length vector of elements, thus $domain(\text{QUEUE}(\text{ELEM})) = domain(\text{ELEM})^*$. In the case of a bounded queue, the length is bounded by the *capacity* Z of the queue.

Formally, the queue interface contains the following terms.

- $empty^t$ and $full^t$ are Boolean formulas that tell whether the queue is empty or full, respectively, at the time step t with $0 \leq t \leq K$. A bounded queue is full iff it contains Z elements. An unbounded queue is never full.
- $firstelem^t$ is of type ELEM and holds the value of the first element in the queue at the time step t. It has a meaningless value if the queue is empty.
- deq^t is a client-controlled Boolean variable that determines whether the first element of the queue is removed when moving to the time step $t + 1$.
- enq^t is a client-controlled Boolean variable that determines whether the element $newelem^t$ is appended to the queue when moving to the time step $t + 1$.
- $newelem^t$ of type ELEM is a client-controlled term, see the previous item.
- $equal^{t,u}$ is a Boolean formula that is true iff the contents of the queue at time steps t and u are the same.

It is assumed that the client of the queue interface never tries to (i) dequeue when the queue is empty, or (ii) enqueue when the queue is full and dequeuing not is taking place at the same time step. That is, the following are assumed to be invariants (i.e. to hold at every time step t):

$$deq^t \Rightarrow \neg empty^t \tag{1}$$

$$enq^t \Rightarrow (\neg full^t \vee deq^t) \tag{2}$$

Formally, the contents of a queue Q evolve in time as follows. Let the contents at the time step t be $Q^t = \langle v_1, v_2, \ldots, v_n \rangle$. Then $firstelem^t = v_1$ and the contents at the next time step are

$$Q^{t+1} = \begin{cases} \langle v_2, \ldots, v_n, newelem^t \rangle & \text{if } \neg empty^t \wedge deq^t \wedge enq^t \\ \langle v_2, \ldots, v_n \rangle & \text{if } \neg empty^t \wedge deq^t \wedge \neg enq^t \\ \langle v_1, \ldots, v_n, newelem^t \rangle & \text{if } \neg deq^t \wedge enq^t \\ \langle v_1, \ldots, v_n \rangle & \text{if } \neg deq^t \wedge \neg enq^t. \end{cases} \tag{3}$$

We point out that the queue interface is not as expressive as a true theory of queues would be. In particular, we cannot define arbitrary relationships between time points, for example, constraining that Q^6 is equal to Q^2 except that the first element has been dequeued. However, from the BMC point of view, arbitrary constraints between queue

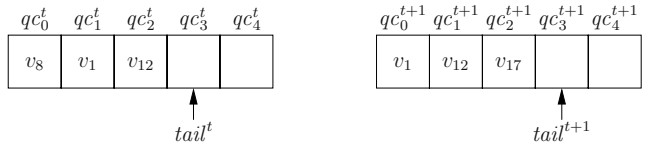


Fig. 1. An illustration of the shifting-based encoding approach when $Q^t = \langle v_8, v_1, v_{12} \rangle$, $deq^t = enq^t = \mathbf{true}$, and $newelem^t = v_{17}$, resulting in $Q^{t+1} = \langle v_1, v_{12}, v_{17} \rangle$

variables are not needed, and it suffices to reason about a non-branching evolution of the contents of the queue. The restrictions on the interface make it possible to design compact encodings to be used especially with BMC.

3 Queue Encodings

In this section we present the three different approaches for encoding queues, called "shifting", "cyclic", and "linear", together with variations within each approach. We analyze the sizes and theory requirements of the alternatives.

3.1 A Shifting-Based Approach

Our first approach, illustrated in Fig. 1, is a straightforward implementation of the BMC semantics of queues given in Eq. (3). It considers bounded queues with at most Z elements and is basically the approach presented in [2] except that only pure FIFO queues are considered here. For each time step t, we introduce a sequence $\langle qc_0^t, \ldots, qc_{Z-1}^t \rangle$ of variables, each of type ELEM. The intuition is that qc_{s-1}^t holds the value of the s:th element in the queue at time step t (the semantics is undefined if there are less than s elements). In addition, we introduce a timed integer variable $tail^t$ that holds the location of the first unused slot in the sequence, i.e. the length of the queue at time step t. Letting s quantify over $\{0, \ldots, Z-1\}$, the definitions for the queue interface variables as well as for updating the queue contents are the following.

$$tail^{t+1} := \begin{cases} deq^t \wedge \neg enq^t & : tail^t - 1 \\ \neg deq^t \wedge enq^t & : tail^t + 1 \\ \mathsf{else} & : tail^t \end{cases} \tag{4}$$

$$empty^t := (tail^t = 0) \tag{5}$$

$$full^t := (tail^t = Z) \tag{6}$$

$$qc_s^{t+1} := \begin{cases} enq^t \wedge \neg deq^t \wedge tail^t{=}s & : newelem^t \\ enq^t \wedge deq^t \wedge tail^t{=}s{+}1 & : newelem^t \\ deq^t & : qc_{s+1}^t \\ \mathsf{else} & : qc_s^t \end{cases} \tag{7}$$

$$firstelem^t := qc_0^t. \tag{8}$$

The notation of Eqs. (4) and (7) should be interpreted as a standard case-expression, i.e. $tail^{t+1} := \mathsf{if}\ deq^t \wedge \neg enq^t\ \mathsf{then}\ tail^t{-}1\ \mathsf{else}\ (\mathsf{if}\ \neg deq^t \wedge enq^t\ \mathsf{then}\ tail^t{+}1\ \mathsf{else}\ tail^t)$.

The term qc_Z^t that appears in (7) in the boundary case $s = Z - 1$ can be taken to have an arbitrary constant value of type ELEM.

As updating the queue contents (Eq. (7)) requires $\mathcal{O}(Z)$ definitions for each time step, the overall size of the encoding with BMC bound K is $\mathcal{O}(K \cdot Z)$.

Equality test. In this approach it is straightforward to check in size $\mathcal{O}(Z)$ whether the contents of the queue are the same at two time steps t and u:

$$equal^{t,u} := (tail^t = tail^u) \wedge \bigwedge_{0 \leq s < Z} (s < tail^t) \Rightarrow (qc_s^t = qc_s^u). \tag{9}$$

One-Hot Encoding for the Tail Pointer. The integer tail pointer used in the shifting-based approach above requires that the SMT solver includes a decision procedure for integers with constants and the successor function. We can eliminate this requirement by using a Boolean one-hot encoding for the tail pointer as follows. For each $s \in \{0, 1, \ldots, Z\}$, introduce a timed Boolean variable $tail_s^t$. In any satisfying truth assignment, at each time step t, exactly one of the variables $tail_0^t, \ldots, tail_Z^t$ will be true by construction. Letting s quantify over $\{0, 1, \ldots, Z\}$, the updating of the tail pointer is expressed as

$$tail_s^{t+1} := \begin{cases} enq^t \wedge \neg deq^t & : (s > 0) \wedge tail_{s-1}^t \\ \neg enq^t \wedge deq^t & : (s < Z) \wedge tail_{s+1}^t \\ \mathsf{else} & : tail_s^t \end{cases} \tag{10}$$

where each $(s > 0)$ and $(s < Z)$ is interpreted as a constant $\mathbf{false}$ or $\mathbf{true}$ depending on s. Equations (5)–(7) are modified by substituting each equality check of the form $tail^t = c$ with the variable $tail_c^t$.

Although updating the tail pointer now requires $\mathcal{O}(Z)$ definitions (Eq. (10)) instead of one as in the integer case (Eq. (4)), the size of the overall encoding stays in $\mathcal{O}(K \cdot Z)$. The potential benefits of the one-hot encoding are that (i) no additional theories are required by the queue encoding, and (ii) as Boolean SAT solvers (whose search techniques and data structures modern SMT solvers apply) are very efficient, the introduced Boolean constraints are possibly easier to solve for SMT solvers.

Equality test. The predicate for checking the equality of queue contents at two time steps is relatively easy to express also in this encoding. Let $equal^{t,u} := equal_0^{t,u}$, where the auxiliary predicate $equal_0^{t,u}$ is defined as

$$equal_s^{t,u} := \left(tail_s^t \wedge tail_s^u\right) \vee \left((qc_s^t = qc_s^u) \wedge equal_{s+1}^{t,u}\right) \quad \text{for } 0 \leq s < Z, \text{ and}$$
$$equal_Z^{t,u} := \left(tail_Z^t \wedge tail_Z^u\right).$$

The size of the $equal^{t,u}$ formula stays the same $\mathcal{O}(Z)$ as in the integer encoded tail pointer case above.

3.2 A Cyclic Approach

We can modify the encoding of Sect. 3.1 by introducing a timed integer variable $head^t$ that tells the position of the first element of the queue. Instead of shifting the entire contents of the queue upon a dequeue operation, we increment $head^t$ by one. This requires

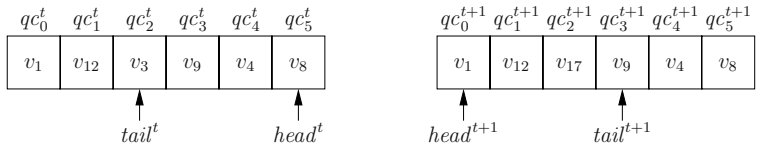


Fig. 2. An illustration of the cyclic encoding approach when $Z = 5$, $Q^t = \langle v_8, v_1, v_{12}\rangle$, $deq^t = enq^t = \mathbf{true}$, and $newelem^t = v_{17}$, resulting in $Q^{t+1} = \langle v_1, v_{12}, v_{17}\rangle$

that the values of $head^t$ and $tail^t$ wrap around at the boundary Z. For notational convenience, we define the terms "successor modulo $Z+1$" and "predecessor modulo $Z+1$" by "$succ(x)$:= if $x = Z$ then 0 else $x+1$" and "$pred(x)$:= if $x = 0$ then Z else $x-1$", respectively. We define

$$head^{t+1} := \text{if } deq^t \text{ then } succ(head^t) \text{ else } head^t \tag{11}$$
$$tail^{t+1} := \text{if } enq^t \text{ then } succ(tail^t) \text{ else } tail^t \tag{12}$$
$$empty^t := (tail^t = head^t) \tag{13}$$
$$full^t := (succ(tail^t) = head^t). \tag{14}$$

There are several approaches for representing the queue contents at each time step, as discussed in the following sub-sections. These vary in compactness and in their requirements for the available decision procedures.

Explicit Contents Representation. As in the shifting-based approach in Sect. 3.1, we introduce a sequence $\langle qc_0^t, \ldots, qc_Z^t\rangle$ of timed variables, each of type ELEM. The queue contents update and the $firstelem^t$ term are written as follows (see Fig. 2).

$$qc_s^{t+1} := \text{if } enq^t \wedge (tail^t{=}s) \text{ then } newelem^t \text{ else } qc_s^t \tag{15}$$

$$firstelem^t := \begin{cases} head^t = 0 & : qc_0^t \\ \quad\vdots & \quad\vdots \\ head^t = Z-1 & : qc_{Z-1}^t \\ \text{else} & : qc_Z^t \end{cases} \tag{16}$$

There are $Z+1$ frame definitions (15), each of constant size. On the other hand, Eq. (16) is of size $\mathcal{O}(Z)$. Thus the size of the overall BMC encoding is $\mathcal{O}(K \cdot Z)$.

Equality test. Perhaps the easiest way of forming the equality predicate $equal^{t,u}$ is to use the one presented below for one-hot encoded head and tail pointers and simply replace each test of the form "$head_c^t$" with "$head^t = c$" and "$tail_c^t$" with "$tail^t = c$".

One-Hot Encoding for Head and Tail. Similarly to Sect. 3.1, we can express the head and tail pointers with Boolean one-hot encoding instead of integers. We only apply this one-hot encoding with the explicit contents representation. For each $s \in \{0, 1, \ldots, Z\}$, introduce timed Boolean variables $head_s^t$ and $tail_s^t$. We replace definitions (11)–(14) with the following for $s = 0, \ldots, Z$.

$$head_s^{t+1} := \text{if } deq^t \text{ then } head_{pred(s)}^t \text{ else } head_s^t \tag{17}$$

$$tail_s^{t+1} := \text{if } enq^t \text{ then } tail_{pred(s)}^t \text{ else } tail_s^t \tag{18}$$

$$empty^t := \bigvee_{0 \le s' \le Z} (head_{s'}^t \wedge tail_{s'}^t) \tag{19}$$

$$full^t := \bigvee_{0 \le s' \le Z} (head_{s'}^t \wedge tail_{pred(s')}^t) \tag{20}$$

Again, the definitions (15) and (16) are modified by substituting each test $head^t = c$ with $head_c^t$ and each $tail^t = c$ with $tail_c^t$.

Compared to the integer case (Eqs. (11)–(14)), maintaining head, tail, empty, and full definitions now requires terms of size $\mathcal{O}(Z)$ instead of $\mathcal{O}(1)$ per time step. However, the total BMC encoding size for K steps stays in $\mathcal{O}(K \cdot Z)$.

Equality test. The predicate for checking the equality of queue contents at two time steps is more cumbersome than in Sect. 3.1 as the head position is now variable. We can define

$$equal^{t,u} := \bigwedge_{0 \le i,j \le Z} (head_i^t \wedge head_j^u) \Rightarrow E_{i,j}^{t,u}, \tag{21}$$

where the $E_{i,j}^{t,u}$ are predicates constrained by

$$E_{i,j}^{t,u} \Leftrightarrow (tail_i^t \wedge tail_j^u) \vee \left(\neg tail_i^t \wedge \neg tail_j^u \wedge (qc_i^t{=}qc_j^u) \wedge E_{succ(i),succ(j)}^{t,u}\right). \tag{22}$$

Intuitively, $E_{i,j}^{t,u}$ is a "suffixes are equal" predicate evaluating to true iff the sequence $\langle qc_i^t, qc_{succ(i)}^t, \ldots, qc_{pred(tail^t)}^t \rangle$ is the same as $\langle qc_j^u, qc_{succ(j)}^u, \ldots, qc_{pred(tail^u)}^u \rangle$. The size of the $equal^{t,u}$ predicate, including the constraints in (22), is $\mathcal{O}(Z^2)$.

UIF-Based Contents Representation. Instead of having a variable qc_s^t to represent the value of the element s at the time step t, we can encode the contents of all elements at a single time step by using an uninterpreted function (UIF). That is, for each time step t, we introduce an UIF $\mathsf{qc}^t : \text{INT} \to \text{ELEM}$ and rewrite the equations (15) and (16) as

$$\mathsf{qc}^{t+1}(s) := \text{if } enq^t \wedge (tail^t{=}s) \text{ then } newelem^t \text{ else } \mathsf{qc}^t(s) \tag{23}$$

$$firstelem^t := \mathsf{qc}^t(head^t) \tag{24}$$

where s ranges over $\{0, ..., Z\}$. The idea is to reduce the size of the definition of $firstelem^t$ from $\mathcal{O}(Z)$ to a constant. However, the overall encoding size still remains in $\mathcal{O}(K \cdot Z)$ as the frame constraints (23) are essentially the same as in the explicit encoding.

Equality test. In the cyclic approach, we can express the length of the queue with "$len^t := \text{if } head^t \le tail^t \text{ then } tail^t - head^t \text{ else } tail^t + Z + 1 - head^t$". Now we can define the queue contents equality checking predicate as

$$equal^{t,u} := (len^t = len^u) \wedge \bigwedge_{0 \le i < Z} ((i < len^t) \Rightarrow E_i^{t,u}) \tag{25}$$

where $E_i^{t,u} := \left(\mathsf{qc}^t(succ^i(head^t)) = \mathsf{qc}^u(succ^i(head^u))\right)$, and $succ^i(x)$ denotes the nested application of the $succ(x)$-notation i times.

Array-Based Contents Representation. We can avoid writing the $Z+1$ copies of the frame constraint (23) by using an array instead of an UIF to represent the queue contents. The downside is the reliance on the more complex theory of arrays (see e.g. [20]). We denote the operations on arrays by $read(a, i)$, which returns the value at index i in array a, and $write(a, i, v)$, which returns a copy of array a in which the value at index i has been replaced by v. We introduce a timed array variable $qc^t : \text{INT} \to \text{ELEM}$ that describes the queue contents at time t. The definitions (15) and (16) are replaced with

$$qc^{t+1} := \mathsf{if}\ enq^t\ \mathsf{then}\ write(qc^t, tail^t, newelem^t)\ \mathsf{else}\ qc^t \tag{26}$$

$$firstelem^t := read(qc^t, head^t). \tag{27}$$

That is, only a constant amount of definitions are needed for each time step, meaning that the encoding is independent of the queue capacity Z and the size of the resulting overall BMC encoding is $\mathcal{O}(K)$.

Equality test. Unfortunately the compactness of the array-based contents representation does not seem to extend to equality checking. The most compact way we have found for expressing $equal^{t,u}$ in this setting is essentially the one for UIF-based contents representation given in Eq. (25). The only change is to replace each equality test of the form $\mathsf{qc}^t(i) = \mathsf{qc}^u(j)$ with $read(qc^t, i) = read(qc^u, j)$.

3.3 A Linear Approach

We next show a very compact encoding approach exploiting uninterpreted functions and a small fragment of linear arithmetic. The resulting encoding has only a *constant* amount of constraints per time step. A drawback is that, like the UIF- and array-based contents representation approaches above, it requires theory combination: if handling of queue elements otherwise requires a decision procedure for a theory T, then the combination of T with the theory of "EUF + integer offsets" (see [21] for an efficient decision procedure for this theory) is required after introducing the queue constraints.

The basic idea, illustrated in Fig. 3, is very simple: we have a *single*, *infinite* array *common to all time steps* in which the queue progresses as a sliding window. For each time step t, we introduce two integer variables, $head^t : \text{INT}$ and $tail^t : \text{INT}$. The contents of the queue at the time step are the array elements from the index $head^t$ to $tail^t - 1$. When an element is removed from the queue, the $head^t$ variable is incremented by one. Similarly, when an element is inserted in the queue, it is written to the array at index $tail^t$, after which the tail index is incremented by one. Thus each array index is *written at most once*. This allows us to capture the contents of the queue by using an UIF $\mathsf{qc} : \text{INT} \to \text{ELEM}$. In contrast to the UIF-based contents encoding in Sect. 3.2, the UIF is not time-dependent but shared across all time steps. We define

$$head^{t+1} := \mathsf{if}\ deq^t\ \mathsf{then}\ head^t + 1\ \mathsf{else}\ head^t \tag{28}$$

$$tail^{t+1} := \mathsf{if}\ enq^t\ \mathsf{then}\ tail^t + 1\ \mathsf{else}\ tail^t \tag{29}$$

$$empty^t := (head^t = tail^t) \tag{30}$$

$$full^t := (tail^t = head^t + Z) \tag{31}$$

$$firstelem^t := \mathsf{qc}(head^t) \tag{32}$$

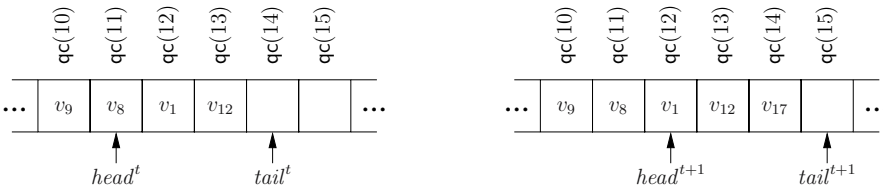


Fig. 3. An illustration of the linear encoding approach when $Q^t = \langle v_8, v_1, v_{12} \rangle$, $deq^t = enq^t = \mathbf{true}$, and $newelem^t = v_{17}$, resulting in $Q^{t+1} = \langle v_1, v_{12}, v_{17} \rangle$

and constrain that

$$enq^t \Rightarrow (\mathsf{qc}(tail^t) = newelem^t). \tag{33}$$

Only a constant amount of terms is needed for each time step, and thus the size of the overall encoding for K time steps is $\mathcal{O}(K)$. The size is the same as that of the bounded cyclic, array-based encoding. The relative benefit is that an easier theory (EUF + integer offsets [21]) is applied and that unbounded queues can be supported (as explained below).

Equality test. The equality checking predicate can be expressed in size $\mathcal{O}(Z)$. Let $len^t := tail^t - head^t$ and define $equal^{t,u} := equal_0^{t,u}$, where

$$equal_s^{t,u} := \left(len^t{=}s \wedge len^u{=}s\right) \vee \left((\mathsf{qc}(head^t{+}s){=}\mathsf{qc}(head^u{+}s)) \wedge equal_{s+1}^{t,u}\right) \tag{34}$$

for $0 \leq s < Z$, and

$$equal_Z^{t,u} := \left(len^t = Z \wedge len^u = Z\right). \tag{35}$$

Unbounded Queues. The linear approach can be modified to allow encoding of *unbounded* queues. Simply replace Eq. (31) above with

$$full^t := \mathbf{false}. \tag{36}$$

The size of the encoding stays in $\mathcal{O}(1)$ per time step.

Equality test. The queue contents equality comparison is similar to that of the bounded case except that the size of the queue at a time step t is now bounded above by $t + M$ instead of the queue capacity Z, where M is the number of elements in queue at the first time step 0. That is, assuming $t < u$, the equality checking predicate $equal^{t,u}$ is the same as in the bounded linear case considered above except that Z is replaced with the constant $t + M$ in Eqs. (34) and (35). The worst case size of $equal^{t,u}$ is thus $\mathcal{O}(K)$. This is a drawback as in BMC for liveness properties we usually have to apply at least K such predicates and thus the overall BMC encoding becomes at least quadratic in the bound.

4 Tag-Based Tuple Element Compression

It is often the case that a queue does not contain scalar values, but *tuples* of values. In the context of UML model checking [3], this happens when the messages in the input

queues of state machines are composed of both a signal identifier and some parameter values associated with the signal. Some SMT solvers, such as Yices [16] and Z3 [22], have a direct support for tuples and thus one can simply define ELEM to be tuple type and use the presented queue encodings unmodified. We next consider solutions for the case that tuples are not supported.

The straightforward way to handle tuple types is to split them into individual parts. For example, if ELEM $=$ TUPLE(REAL, INT), then an uninterpreted function qc^t : INT $\rightarrow$ ELEM appearing in the UIF-based contents representation scheme (Sect. 3.2) is encoded as two UIFs qc_1^t : INT $\rightarrow$ REAL and qc_2^t : INT $\rightarrow$ INT, and a constraint $\mathsf{qc}^t(x) = \mathsf{qc}^t(y)$ becomes $\mathsf{qc}_1^t(x){=}\mathsf{qc}_1^t(y) \wedge \mathsf{qc}_2^t(x){=}\mathsf{qc}_2^t(y)$. This transformation can be applied to all queue encodings of Sect. 3. If the element type ELEM is a tuple with A parts, then the variables $\mathit{firstelem}^t$, $\mathit{newelem}^t$, qc_s^t, and qc^t, and the UIFs qc^t and qc need to be duplicated A times together with the constraints involving those variables and UIFs. This increases the sizes of all encodings by a factor of A in the $\mathcal{O}$-notation. We will call the result the *duplicating* tuple encoding.

The alternative we propose is a *tag-based* encoding that avoids storing tuple values in the queue and moving them across time steps. Instead, each enqueued tuple is associated with a *tag*, e.g. a single integer value, which is stored in the queue. Upon dequeuing, the tag is decoded back into a tuple. The scheme can be efficiently implemented using UIFs as follows. Assume that ELEM $=$ TUPLE$(T_1, \ldots, T_A)$ for some types T_i. We define a scalar type TAG that has to have a domain large enough to hold the possible element values, i.e. $|\mathit{domain}(\text{TAG})| \geq |\mathit{domain}(\text{ELEM})|$ should hold. We define time-independent UIFs decode_i : TAG $\rightarrow T_i$ for each $1 \leq i \leq A$ that are used to interpret the tags as tuple parts, and construct a queue with element type TAG using one of the encodings presented in the previous section. We rename the terms $\mathit{firstelem}^t$ and $\mathit{newelem}^t$ of the interface of the queue as $\mathit{firsttag}^t$ and newtag^t, respectively, and hide them from the client. Instead, the client will see the terms

$$\mathit{firstelem}_i^t := \mathsf{decode}_i(\mathit{firsttag}^t) \tag{37}$$

$$\mathit{newelem}_i^t := \mathsf{decode}_i(\mathit{newtag}^t) \tag{38}$$

for each $1 \leq i \leq A$ as part of the queue interface. Except for the queue equality predicate $\mathit{equal}^{t,u}$ discussed below, this additional level of abstraction does not affect the semantics of the queue. Note that only the decode functions together with the definitions (37) and (38) are duplicated A times, while the internals of the queue only deal with scalar values. Thus when the tag-based encoding is applied, the size of the shifting-based approach as well as that of the cyclic approach with explicit and UIF-based contents representation drop from $\mathcal{O}(K \cdot Z \cdot A)$ to $\mathcal{O}(K \cdot (Z + A))$. The size of the array-based contents representation stays in $\mathcal{O}(K \cdot A)$ but requires only one array variable instead of A per time step; this is, in theory, beneficial as the theory of equality with UIFs required by tags is much easier to decide than the theory of arrays.

As tuples with same values can be assigned to different tags, the equality checking predicate $\mathit{equal}^{t,u}$ needs special treatment. In $\mathit{equal}^{t,u}$, every equality comparison between tag values has to be expanded; for instance, the comparison $qc_s^t = qc_s^u$ in Eq. (9) has to be rewritten as $\bigwedge_{1 \leq i \leq A} (\mathsf{decode}_i(qc_s^t) = \mathsf{decode}_i(qc_s^u))$. Thus for the equality checking part of the encoding, tags do not help to compress the size of the encoding.

Table 1. Comparison of the approaches on the single queue stress test with a scalar integer element. The numbers show the largest bound that was solved within ten minutes by the Z3 solver.

	shifting		cyclic				linear	unbounded
			explicit		uif	array		
Z	int. tail	one-hot tail	int. tail	one-hot tail				
2	28	79	35	**92**	24	26	13	12
5	20	33	28	**42**	15	26	12	
10	19	25	20	**31**	14	27	12	
50	19	25	17	**31**	12	25	11	

5 Experiments

We now provide an experimental comparison of the proposed approaches. We have developed a prototyping tool called PySMT for constructing and solving SMT problems in the Python programming language. It (i) has an API for constructing the problems, (ii) can translate the problems into the native input language of several SMT solvers and also (to some extent) to the SMT-LIB format, and (iii) can also execute the solver binary and (to a quite limited extent) parse the result so that it can be queried by using the API. We have implemented the proposed queue encoding approaches on top of PySMT and give some preliminary experimental results below. The scripts and models for the experiments are available at `http://www.tcs.hut.fi/~tjunttil/experiments/LPAR2008-SMT`.

5.1 Single Queue Stress Test

First, we try to test the efficiency of the queue encodings in isolation by constructing very simple BMC problems consisting of one queue only. In this problem, (i) whether an enqueue or a dequeue operation is applied at time step t is unconstrained, meaning that all possible enqueue/dequeue sequences are considered, (ii) each enqueued element is either an integer or a tuple of integers, each constrained to have a value greater than some positive constant, and (iii) the (valid) property to be checked is that an element with a negative integer value is never dequeued. For each time step, we set the run time limit to ten minutes and report the time spent in the solver. Problem generation time is not included as our generator script has, we believe, unessential inefficiencies.

Scalar Elements. To isolate the core queue constraints from the constraints needed to represent tuple elements, we first compare the encoding approaches in the case of scalar elements. Table 1 shows the results for different queue lengths when Z3 (version 1.2) [22] is used as the SMT solver. In addition, Fig. 4(a) shows a more detailed view when the queue size Z is five. The results show that there are dramatic differences in the performance of different approaches and contents encoding schemes. Unfortunately, the more compact and elegant ones, namely the linear approach and the cyclic approach with array-based contents representation, are not performing well. Instead, the encodings applying fewest theories, i.e. the cyclic and shifting approaches with one-hot

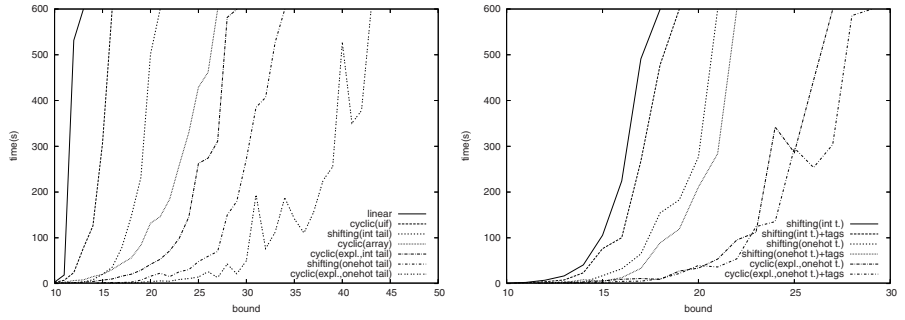
encoded head and tail pointers seem to be the best choices in general. We also ran experiments with the Yices SMT solver [16] and obtained similar results, except that the integer encoded head and tail pointers seem to perform as well as the one-hot encoded.

Tuple Elements. We also benchmarked the encoding approaches when queue elements are tuples of integers. As expected, the result is that the problems become harder when tuples contain more parts. For instance, consider the cyclic approach with explicit contents representation and one-hot encoded head and tail pointers. With scalar element (tuple with one part) the behavior is shown as the rightmost curve in 4(a) while the second rightmost curve in Fig. 4(b) shows the same with a tuple of five parts: the largest bound solved within ten minutes drops from 42 to 27.

The second observation is that the tag-based tuple encoding is almost universally a few times more efficient in terms of running time than the duplicating tuple encoding. Figure 4(b) shows a comparison of the tuple encodings for three different queue encodings approaches; these plots represent typical behavior in this benchmark set. We also experimented with the direct tuple type support of Z3; it seems to provide similar performance as our tag-based encoding. Again, comparable results were obtained when Yices was used as the solver instead of Z3.

5.2 Bounded Model Checking of UML Models

We have also benchmarked the queue encodings in a more realistic bounded model checking context. We analyzed some UML models by using the symbolic encoding described in [2,3]. Instead of using NuSMV, we translate the BMC problems into SMT problems and use Yices (version 1.0.11) [16] to solve them. The results are shown in Table 2, the numbers give the cumulative time (in seconds) used by the SMT solver when solving all the problems from bound 0 to the bound $|\text{cex}|$ where a counter-example to the analyzed property is found. The queue size for the bounded queue encodings was set to ten; "dstep" ("interl.", resp.) denotes that the dynamic step (interleaving, resp.) semantics (see [3]) was applied. The use of tags to represent tuple queue elements seems



(a) (b)

Fig. 4. Comparison of some approaches with Z3 as the solver. (a) Scalar queue element, $Z = 5$. (b) Tuple queue element with 5 parts, $Z = 5$.

Table 2. BMC of some UML models using Yices as the solver

model	|cex|	linear unbounded	shifting	shifting one-hot	cyclic explicit	cyclic one-hot	cyclic UIF
`giop`, dstep	8	21.31	23.93	28.99	296.77	78.98	137.33
(with tags)			24.18	24.52	29.58	25.67	24.44
`giop`, interl.	14	1986.07	1599.10	1250.13	>1h	>1h	2902.11
(with tags)			1084.25	860.11	849.31	1244.03	1088.80
`travel`, interl.	15	1.86	2.54	2.18	3.16	2.93	3.17
(with tags)			2.45	2.09	2.87	2.83	3.26
`mtravel`, dstep	11	3.77	5.43	4.27	7.45	5.74	7.02
(with tags)			3.99	3.79	4.58	4.32	4.82

to play a much bigger role than the encoding approach in this practical setting; they provide a substantial performance gain especially when analyzing the `giop` model having tens of message parameters and thus wide tuples in queues. With the other models having no or only few parameters, the performance gain is non-existent or small; in addition, the choice of the encoding approach does not seem to make much difference on these models. The reason for this is probably that the applied bounds are relatively small and parts other than the queue encoding dominate the search space of the SMT problem.

6 Conclusions

We have presented and experimentally evaluated different quantifier-free SMT encodings for queues in the context of bounded model checking. The presented encodings vary significantly in compactness and the theories they require the SMT solver to implement. Our preliminary experimental results show that the most compact encodings do not necessarily perform best, even when they involve no complex theories such arrays but only equality with uninterpreted functions and integer offsets. On the contrary, it seems that it may be worthwhile to use more space by booleanizing integer head and tail pointers so that the encoding becomes essentially propositional, the only theory atoms being equality tests between elements that are stored in the queue. The proposed method for compressing tuple elements with the use of tags and uninterpreted decode functions yields a relatively consistent and often significant speed-up in our experiments. The most obvious future work is of course to develop and implement decision procedures for theories of queues. The encoding approaches presented in this paper form a natural base when evaluating their performance in the BMC context.

References

1. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
2. Dubrovin, J., Junttila, T.: Symbolic model checking of hierarchical UML state machines. In: ACSD 2008, pp. 108–117. IEEE Press, Los Alamitos (2008)

3. Dubrovin, J., Junttila, T., Heljanko, K.: Symbolic step encodings for object based communicating state machines. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 96–112. Springer, Heidelberg (2008)
4. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV version 2: An opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
5. de Moura, L.M., Dutertre, B., Shankar, N.: A tutorial on satisfiability modulo theories. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 20–36. Springer, Heidelberg (2007)
6. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). Journal of the ACM 53(6), 937–977 (2006)
7. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T.A., van Rossum, P., Schulz, S., Sebastiani, R.: MathSAT: Tight integration of SAT and mathematical decision procedures. Journal of Automated Reasoning 35(1–3), 265–293 (2005)
8. Bjørner, N.S.: Integrating Decision procedures for Temporal Verification. PhD thesis, Stanford University (1998)
9. OMG: UML 2.0 superstructure specification (2005), `http://www.omg.org`
10. International Telecommunication Union Geneva, Switzerland: Recommendation Z.100 (03/93) - CCITT specification and description language (SDL) (1993)
11. Biere, A., Heljanko, K., Junttila, T.A., Latvala, T., Schuppan, V.: Linear encodings of bounded LTL model checking. Logical Methods in Computer Science 2(5) (2006)
12. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Johnson, S.D., Hunt Jr., W.A. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 108–125. Springer, Heidelberg (2000)
13. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. In: BMC 2003. Electronic Notes in Theoretical Computer Science, vol. 89, pp. 541–638. Elsevier, Amsterdam (2003)
14. Oppen, D.C.: Reasoning about recursively defined data structures. Journal of the ACM 27(3), 403–411 (1980)
15. Lahiri, S.K., Seshia, S.A., Bryant, R.E.: Modeling and verification of out-of-order microprocessors in UCLID. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 142–159. Springer, Heidelberg (2002)
16. Dutertre, B.: System description: Yices 1.0.10. SMT-COMP 2007 tool description paper (2007), `http://www.smtcomp.org/2007/participants.shtml`
17. Ganai, M.K., Gupta, A., Ashar, P.: Efficient modeling of embedded memories in bounded model checking. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 440–452. Springer, Heidelberg (2004)
18. Ganai, M.K., Gupta, A., Ashar, P.: Verification of embedded memory systems using efficient memory modeling. In: DATE 2005, pp. 1096–1101. IEEE Computer Society, Los Alamitos (2005)
19. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (1999)
20. Stump, A., Barrett, C.W., Dill, D.L., Levitt, J.R.: A decision procedure for an extensional theory of arrays. In: LICS 2001, pp. 29–37. IEEE Computer Society, Los Alamitos (2001)
21. Nieuwenhuis, R., Oliveras, A.: Fast congruence closure and extensions. Information and Computation 205, 557–580 (2007)
22. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)