Program Verification as Satisfiability Modulo Theories

Nikolaj Bjørner
Microsoft Research

Ken McMillan
Microsoft Research

Andrey Rybalchenko
Technische Universität München

Abstract

A key driver of SMT over the past decade has been an interchange format, SMT-LIB, and a growing set of benchmarks sharing this common format. SMT-LIB captures very well an interface that is suitable for many tasks that reduce to solving first-order formulas modulo theories. Here we propose to extend these benefits into the domain of symbolic software model checking. We make a case that SMT-LIB can be used, and to a limited extent adapted, for exchanging symbolic software model checking benchmarks. We believe this layer facilitates dividing innovations in modeling, developing program logics and front-ends, from developing algorithms for solving constraints over recursive predicates.

1 SMT and checking properties of recursive predicates

Progress in modern SMT solvers has been driven by a large set of applications in diverse areas. Applications reduce problems to common first-order formats, SMT-LIB [5] and now also the THF [13] format in TPTP, for exchanging benchmarks. They benefit from the progress in automated deduction in SMT solvers and utilize the smorgasbord of theories that are supported with efficient and dedicated solvers. Common to SMT solvers is their support for simply typed first-order classical logic augmented with theories. They solve *satisfiability* of formulas over the supported language. When SMT solvers are used to discharge verification conditions from program verifiers, it is important to be able to check validity, or dually unsatisfiability. When SMT solvers are used in dynamic symbolic simulation, the more interesting question is to determine satisfiability and find a satisfying assignment to formulas. A key driver of progress in these applications has been the ability to exchange problems using standardized and well-defined logics, theories and syntax, especially as embodied in the SMT-LIB standard.

Here we propose to extend these benefits into the domain of symbolic software model checking. This problem reduces to the problem of inference of the intermediate specifications required by various proof systems. These specifications can be, for example, loop invariants, procedure summaries, environment assumptions or dependent types. As observed in [6], the inference problem in turn reduces to the problem of satisfiability of a certain class of first-order constraints containing unknown relations. These constraints can be produced by existing verification condition generators, which already typically target the SMT-LIB standard. We will argue that, with very slight extensions, SMT-LIB can be adapted to be a standard interchange medium for software model checking and related problems. The advantage of such an approach is a clean separation of concerns: the interpretation of programming language is left to verification condition generators, while model checking is handled by purely logic-based tools. This separation could have several benefits. It relieves the algorithm implementer from the need to deal with the complexity of programming languages, it allows implementations to be easily re-targeted to different languages, and it allows algorithms to be compared directly, without concern for the ambiguity of how programming languages are modeled. At the very least it can be instrumental for comparing algorithms for checking recursive predicates whether they come from software model checking or not.

In this paper, we propose extensions to SMT-LIB to allow it to be used as an interchange format for software model checking benchmarks. We are not alone in suggesting ways of leveraging SMT-LIB for verifying reactive systems and software. The VMT [1] effort, see also the proceedings for this workshop (SMT 2012), proposes extending SMT-LIB with notation for transitions. Our suggestion is to utilize Horn clauses as a low-level, but uniform, exchange format. Transition systems and, more generally, push-down systems can be translated into Horn clauses. Of course, Horn clauses can be translated into transition systems by using a theory that can express stacks. The Numerical Transition Systems Library also aims to identify an interchange format for transition systems. We aim to provide a basis for comparing various tools that work on various incarnations of recursive predicates, such as [14, 2, 3, 8, 1].

2 Program verification Tool work-flow

A proof system gives us rules for decomposing the proof of a complex goal into proofs of simpler sub-goals. These rules are typically non-deterministic, requiring us to instantiate some auxiliary construct. Consider, for example, the proof rule for sequential compositions in Hoare logic:

$$\frac{\{P\} \;\; s_1 \;\; \{Q\} \;\; s_2 \;\; \{R\}}{\{P\} \;\; s_1; s_2 \;\; \{R\}}$$

Here, the intermediate assertion Q is the auxiliary construct that must be instantiated in order to decompose the proof. The decomposition yields two sub-goals above the line, one involving s_1 and the other involving s_2. The difficulty in this step is to infer the predicate Q.

In software model checking we are concerned with the problem of automatically *inferring* these auxiliary proof constructs. Here we would like to argue that this inference problem can be handled in a generic way. That is, we are interested in methods that are to some extent independent of the proof system, and require only that the proof sub-goals can be expressed in a suitable logical form.

In program proving, the translation of proof sub-goals into logic is commonly referred to as verification condition (VC) generation. For example, suppose in the above example that our program has one variable x. The VC's can be written in this form:

$$\begin{array}{lcl} \forall x, x'. \; (P(x) \wedge [s_1](x, x') & \Rightarrow & Q(x')) \\ \forall x, x'. \; (Q(x) \wedge [s_2](x, x') & \Rightarrow & R(x')) \end{array}$$

where $[s_i]$ is some relation representing the operational semantics of program statement s_i. If these two formulas are shown to be valid, then our proof sub-goals are discharged, and we know that $\{P\}$ $s_1; s_2$ $\{R\}$ holds. Notice though, that these VC's contain one unknown relation Q that must be inferred. In a typical VC checking application, the user would provide the relation Q in logical form, and the task of the verification tool is to check that the two individual proof sub-goals are valid. On the other hand, the inference problem is to *simultaneously* solve the VC's for a value of Q that makes both true. Both of these problems are satisfiability problems. In the former case, we separately check satisfiability of the negation of each condition, with a fixed value for Q, while in the latter, we check satisfiability of both constraints together, with an uninterpreted symbol standing in for Q.

We will observe that the VC's for a wide variety of proof systems have a particular form. That is, they are of the form $\forall X. \; (B \Rightarrow H)$, where X is some set of first-order variables, H is

[1] `https://sites.google.com/site/torino2011ic0901/programme/talks`

an application of some (possibly unknown) relation, and B is a formula in which the unknown relations occur only positively (that is, under an even number of negations) and not under quantifiers. For example, the two VC's above are of this form. We will refer to such formulas as generalized Horn formulas. We will say that H is the *head* of the generalized Horn formula, B is the *body*, and the *arity* of a generalized Horn formula is the number of occurrences of an unknown relation in the body.

Quite a variety of proof systems produce VC's as generalized Horn formulas. For example, within procedures, Hoare logic produces sub-goals of the form $\{P\} \;\; s \;\; \{Q\}$, which as we have seen yield generalized Horn formulas of arity one as VC's.

Now let's consider procedure calls, in the way they are handled by the verification condition generator for the Boogie programming language [4]. A parameter-less procedure π in Boogie has the following form:

```
procedure $\pi$:
    requires $P(X)$;
    ensures $Q(X_{old}, X)$;
    $\sigma$
```

Here, formula P is a required precondition to enter the procedure, formula Q is the guaranteed post-condition and statement σ is the body of the procedure. The precondition is over the program variables X, while the post-condition refers to the program variables X and their entry procedure entry values X_{old}.

To generate verification conditions, Boogie replaces each call to π with the following code:

```
    assert $P(X)$;
    $X_{old} := X$;
    havoc $X$;
    assume $Q(X_{old}, X)$;
```

The havoc statement causes the variables X to take a non-deterministic value. Thus, the transformed call to π behaves as any code satisfying the specification of π. Now suppose the π is called in the following context:

```
    assert $R(X)$;
    call $\pi$;
    assert $S(X)$;
```

After replacing the call to π, the verification conditions we obtain from the weakest precondition calculus are:

$$\begin{array}{rcl} \forall X.\; R(X) & \Rightarrow & P(X) \\ \forall X_{old}, X.\; (R(X_{old}) \wedge Q(X_{old}, X) & \Rightarrow & S(X)) \end{array}$$

Note that these formulas are also in generalized Horn formula form and that the second VC has arity two. That is, there are two unknown relations in the body of the second VC.

A very simple modular proof system for concurrent programs is given in [7]. We are given a collection of N parallel process of the form:

```
process $\pi_i$:
    while * do
        $\rho_i$
```

Each process π_i has a set of local variables X_i. The processes share a set Y of global variables. We wish to prove in a modular way that, given some initial condition $\phi_{init}(X_1, \ldots, X_N, Y)$, some error condition $\phi_{error}(X_1, \ldots, X_N, Y)$ is never true. The relations to be inferred for each process i are $R_i(X_i, Y)$, which is the invariant of the loop, and $E_i(Y, Y')$, which is the environment constraint. The verification conditions for process π_i are:

$$\begin{array}{rcl}
\forall X_1, \ldots, X_N, Y.\ \phi_{init}(X_1, \ldots, X_N, Y) & \Rightarrow & R_i(X_i, Y) \\
\forall X_i, Y, X_i', Y'.\ R_i(X_i, Y) \wedge [\rho_i](X_i, Y, X_i', Y') & \Rightarrow & R_i(X_i', Y') \\
\forall X_i, Y, Y'.\ R_i(X_i, Y) \wedge E_i(Y, Y') & \Rightarrow & R_i(X_i, Y') \\
\forall X_i, Y, X_i', Y'.\ R_i(X_i, Y) \wedge [\rho_i](X_i, Y, X_i', Y') & \Rightarrow & E_j(Y, Y') \quad \text{for } j \in \{1, \ldots, N\} \setminus \{i\}
\end{array}$$

These conditions state respectively that the initial condition implies the loop invariant, the loop body preserves the loop invariant, environment transitions preserve the loop invariant, and the loop body (executed atomically) is an allowed environment transition for other processes. The global correctness condition is:

$$\forall X_1, \ldots, X_N, Y.\ R_1(X_1, Y) \wedge \cdots \wedge R_N(X_N, Y) \wedge \phi_{error}(X_1, \ldots, X_N, Y) \quad \Rightarrow \quad \textsc{False}$$

Note again that all these proof sub-goals are generalized Horn formulas.

Another example is type checking of dependently typed functional programs. Consider the Liquid Types system [12]. In this system, a *refinement type* has the form $\{\nu : \beta \mid P(\nu, X)\}$, where β is a base type and P is a relation over the value ν and program variables X. For example, $\{\nu : int \mid \nu > x\}$ is the type of integers greater than variable x. A type binding $y : \{\nu : \beta \mid P(\nu, X)\}$ is translated to the logical constraint $[\![P(y, X)]\!]$, where $[\![\cdot]\!]$ is a conservative embedding of program terms into the logic. The typing rules of Liquid Types produce type checking sub-goals of the form $\Gamma \vdash \{\nu : \beta \mid P_1(\nu, X)\} \prec \{\nu : \beta \mid P_2(\nu, X)\}$ where $\prec$ is the subtype relation and Γ is an environment consisting of type bindings and constraints on variables deriving from program guards. Logical verification conditions in the form of generalized Horn formulas are produced by the following rule:

$$\frac{[\![\Gamma]\!] \wedge [\![P_1(\nu, X)]\!] \Rightarrow [\![P_2(\nu, X)]\!]}{\Gamma \vdash \{\nu : \beta \mid P_1(\nu, X)\} \prec \{\nu : \beta \mid P_2(\nu, X)\}}$$

Existence of a relational interpretation satisfying these conditions is a sufficient condition for type safety of the program (though not a necessary condition, since the typing rules are conservative).

3 SMT as a Logical Layer

In the previous section we described how program verification tools produce verification conditions that can be viewed as a conjunction of Horn clauses. Program correctness reduces to finding a solution to the recursive predicates defined by the Horn clauses. We will here elaborate on what this problem means in terms of logical formulas.

3.1 From Recursive Predicates to SMT

In its basic form, the problem posed by solving verification conditions is to determine whether there are solutions to recursive predicates, such that a conjunction of universally quantified Horn formulas is true. In other words, the query is whether a conjunction of formulas is *satisfiable*.

This characterization side-steps the semantics of *auxiliary functions, background theories and assertions*, other than theories that are indigenous to SMT such as arithmetic. We claim that the appropriate semantics is to determine that for every interpretation of a given background theory there is a solution to the recursive predicates. Using logical notation, we check the truth values of formulas of the form:

$$\forall \vec{R}, \vec{f} \ . \ \mathit{Background}[\vec{R}, \vec{f}] \ \Rightarrow \ \exists \vec{P} \ . \ \bigwedge \forall \vec{X} \ . \left(P_1(X_1) \wedge \ldots \wedge P_N(X_N) \wedge \phi[\vec{R}, \vec{f}, \vec{X}] \ \Rightarrow \ P(X) \right) \quad (1)$$

This format is not directly supported by SMT-LIB. It includes universal quantification over relations and functions. The quantifier alternation is also inconvenient if we wish to refer to recursive predicates as free variables (recall that $\exists P.\varphi[P]$ is satisfiable if and only if $\varphi[P]$ is satisfiable, so it is typically more convenient to define P as an uninterpreted predicate). We can change the alternation of quantifiers by exchanging the two outer alternations:

$$\exists \vec{P} \ . \ \forall \vec{R}, \vec{f} \ . \ \mathit{Background}[\vec{R}, \vec{f}] \ \Rightarrow \ \bigwedge \forall \vec{X} \ . \left(P_1(X_1, \vec{R}, \vec{f}) \wedge \ldots \wedge P_N(X_N, \vec{R}, \vec{f}) \wedge \phi[\vec{R}, \vec{f}, \vec{X}] \ \Rightarrow \ P(X, \vec{R}, \vec{f}) \right) \quad (2)$$

The universal quantification over relations and functions is still present. Furthermore, P has additional arguments. Can we use SMT-LIB for higher-order problems? The theory of extensional arrays in SMT-LIB includes the function `select`. It corresponds to function application. This suggests the following approach: treat functions and relations as arrays, then use SMT solvers to check satisfiability of the resulting formula with arrays. The catch is that the model theory for this first-order reformulation does not correspond to the original second-order formulation. In the following, we will avoid these transformations and side-step having to deal with what are potentially adequate (higher-order or Henkin) theories for arrays. Instead we will augment SMT-LIB slightly.

3.2 From SMT to Solutions of Recursive Predicates

Many applications of SMT solvers are only interested in checking validity of formulas. So, given a formula φ, they assert $\neg\varphi$ and check if the SMT solver produces the answer `unsat`. Similar, in program verification, we are interested in checking a property. The most basic answer we would expect from a verification tool for programs is whether the property φ can be established. Dually, we can ask whether the property $\neg\varphi$ is unreachable.

More informative answers are possible. One may want a certificate for unreachability or a certificate for reachability. A certificate for reachability is in a nutshell a proof of unsatisfiability. The proof can be presented as a trace. A certificate for unreachability is in a nutshell a model for the recursive predicates. So what is a model? Saturation based theorem provers may be able to produce a set of saturated clauses that do not include the empty clause and such that any clause that can be deduced from the saturated set is subsumed. Checking that a set of first-order clauses is saturated is relatively straight-forward. Unfortunately, we are not aware of reasonable notions of saturation modulo theories. A different way to represent models is to produce a formula for each recursive predicate. The free variables in the produced formulas correspond to the arguments of the recursive predicate. So in a sense the formulas correspond to macro expansions of the recursive predicates. Checking such a solution requires theorem proving: Plug in the macro expansion for each of recursive predicates and check that the resulting formula is valid. The proof obligations are ground when the background is empty. Existing tools for generating interpolants and lifting PDR to theories [2] do produce such certificates that can be

[2] In the case of PDR, we give a few small examples of such certificates on `http://rise4fun.com/Z3Py/tutorial/fixedpoints`

checked by SMT solvers for ground formulas. We speculate that finer-graned certificates are of potential useful: macros together with proof hints that are used for checking that the macros are indeed solutions.

3.3 Bloat in Translation?

What is the cost of resorting to a logical encoding of programs and their control-flow graphs? We alluded to that a control flow graph (CFG) corresponds directly to a set of Horn clauses. Each node in the CFG is a predicate. Horn clauses encode Hoare triples between locations. In our experience, the main source of bloat is how variable scoping is handled: all local variables in scope have to be either directly arguments of the recursive predicates, or indirectly by using records or arrays that summarize variables. Current tools for solving predicates work with variables enumerated directly. Using records would require additions to SMT-LIB from its current form. In particular, it would be useful to have a way to update a record. One could also consider using arrays instead of records, but general purpose decision procedures for the theory of extensional arrays are not really well tuned for this special purpose usage.

3.4 Loss in Translation?

A dual concern is whether this layer opens up for loss in translation. We argue not, as rules can be in one-to-one correspondence with CFG/abstract syntax tree representations of programs.

3.5 Dealing with Heap

There are many possible ways to model and reason about heaps. Front-end tools may choose a model of their choice and encode the model in background axioms. Back-end solvers are also left open to detect heap operations and provide different abstractions, such as separation logics. Hence, it is also an area where solvers can compete on applying different abstractions. In summary, the approach (with encoding CFGs to SMT constraints) does not by itself require fixing one particular model of heaps.

4 Recursive predicates in Z3

ARMC [11] and HSF [6] encode recursive predicates as Horn formulas in Prolog syntax. In a similar style, μZ [10, 9] uses an extension of the `SMT-LIB` format for representing recursive predicates. Recursive predicates are treated differently from non-recursive predicates. Recursive predicates are defined in rules and checked in queries. Non-recursive predicates and functions may be used in rules and queries; they can be further constrained by background assertions.

4.1 Recursive predicates

Z3 has four syntactic constructs for handling recursive predicates. We summarize these below.

`(rule` *`rule`*`)` declares a rule, where *`rule`* is an SMT-LIB expression expected to be of the form:

```
rule ::= (forall (bound) rule) | (=> body rule) | (R args)
```

The relation *`R`* in the head is expected to be declared as a recursive relation. The supported format for *`body`* is as a quantifier-free formula such that the recursive predicates (such as *`R`*) occur positively.

While the internal engine in μZ uses Horn clauses, rules are not required to be Horn clauses. Nested disjunctions in bodies are eliminated using a Tseitsin style transformation that produces a set of Horn clauses.

`(query` *`query`*`)` where *`query`* `::= (exists (`*`bound`*`)` *`query`*`) |` *`body`*.

`(declare-rel` *`R`* `(`S_1 `...` S_n`))` declares a recursive relation R. It takes n arguments of sort $S_1 \ldots S_n$ respectively.

`(declare-var` *`v`* *`S`*`)` Rules and queries may reference variables declared using `declare-var`. When used in rules, the variables are universally quantified, when used in queries, the variables are existentially quantified. The `declare-var` construct is purely syntactic sugar. It makes writing rules more convenient because it saves the overhead of binding the same variables in different rules.

A query is satisfiable if there are instances of the existential variables and a derivation of recursive rules that derives the query. Otherwise a query is unsatisfiable. This definition corresponds to checking queries with Prolog and Datalog. It is dual to how the semantics corresponds to SMT: when there is no instantiation of variables that satisfy a query it means that there is an interpretation of the recursive predicates that satisfy each rule, but satisfies the negation of the query. On the other hand, if a query has a solution, then the trace that corresponds to deriving the query is a proof: The trace provides quantifier instantiations of the Horn clauses and $\forall \vec{x} \,.\, \neg\texttt{query}$ such that the resulting set of ground Horn clauses is unsatisfiable.

Equation (3) summarizes this characterization. The formula in (3) is closed and the goal is to check if it is equivalent to *true*. The set `rules` contains the recursive Horn clauses defined by `rule` and `query` is a formula corresponding to a query, $\vec{x}$ are declared as variables and $\vec{P}$ are declared as recursive predicates:

$$\exists \vec{P} \,.\, \left(\bigwedge_{r\in\texttt{rules}} \forall \vec{x} \,.\, r(\vec{x})\right) \wedge \forall \vec{x} \,.\, \neg\texttt{query} \tag{3}$$

Example 1. *Suppose we are given a recursive program* `sum`

```
let rec sum n = if n <= 0 then 0 else n + sum (n-1)
```

and we wish to establish that `sum n >= n` *for every* `n`. *The input-output specification of* `sum` *is a relation with one argument for the input and one argument for the output. We can specify* `sum` *using a set of Horn rules and the assertion as a query:*

```
(declare-rel sum (Int Int))
(declare-var n Int)
(declare-var m Int)
(rule (=> (<= n 0) (sum n 0)))
(rule (=> (and (> n 0) (sum (- n 1) m)) (sum n (+ m n))))
(query (and (sum n m) (< m n)))
```

An interpretation for `sum` *that satisfies the Horn clauses and the query is:*

```
(define-fun sum ((A Int) (B Int)) Bool (and (>= B 0) (<= (+ (* (- 1) B) A) 0)))
```

4.2 Background

While currently unsupported in μZ, the format admits free functions and non-recursive predicates. It inter-operates with assertions over non-recursive predicates and functions.

Example 2. *One can assert that a free function* `f` *is injective with partial converse* `g` *using an axiom:*

```
(declare-fun f (Int) Int)
(declare-fun g (Int) Int)
(assert (forall ((x Int)) (= (g (f x)) x)))
```

Then `f` *can be used in rules. For example,*

```
(declare-rel R (Int Int))
(declare-var x Int)
(declare-var y Int)
(rule (R 0 1))
(rule (=> (R x y) (R (f x) (f y))))
(query (and (R x y) (= x y)))
```

The query is unsatisfiable with the justification

```
(define-fun R ((A Int) (B Int)) Bool (not (= A B)))
```

Free functions and non-recursive predicates are treated logically different from recursive predicates. Their meaning is succinctly captured by extending (3) by taking background assertions and auxiliary predicates and functions into account. Let $\vec{R}, \vec{f}$ be the non-recursive predicates and functions, and `asrts` be the current assertions. Correspondence (4), derived from (1), characterizes the semantics in the context of auxiliary predicates, functions and assertions:

$$\forall \vec{R}, \vec{f} \;.\; \texttt{asrts} \;\Rightarrow\; \exists \vec{P} \;.\; \left(\bigwedge_{r \in \texttt{rules}} \forall \vec{x} \;.\; r(\vec{x}) \right) \;\wedge\; \forall \vec{x} \;.\; \neg \texttt{query} \tag{4}$$

5 Summary

We made a case that SMT-LIB is a superb basis for exchanging benchmarks among symbolic software model checkers. As a side-effect we also introduced an approach to specify recursive relations modulo background assertions in the context of SMT-LIB. The format can be used in its own right. The proposed use of SMT-LIB uses a modest extension for describing rules and declaring recursive predicates.

References

[1] Akash Lal and Shaz Qadeer and Shuvendu Lahiri. Corral: A Solver for Reachability Modulo Theories. In *CAV*, July 2012.

[2] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. From under-approximations to over-approximations and back. In *TACAS*, pages 157–172, 2012.

[3] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Whale: An interpolation-based algorithm for inter-procedural verification. In Viktor Kuncak and Andrey Rybalchenko, editors, *VMCAI*, volume 7148 of *Lecture Notes in Computer Science*, pages 39–55. Springer, 2012.

[4] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO*, pages 364–387, 2005.

[5] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[6] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, 2012.

[7] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *POPL*, pages 331–344, 2011.

[8] William R. Harris, Sriram Sankaranarayanan, Franjo Ivancic, and Aarti Gupta. Program analysis via satisfiability modulo path programs. In *POPL*, pages 71–82, 2010.

[9] Kryštof Hoder and Nikolaj Bjørner. Generalized Property Directed Reachability. In *SAT*, 2012.

[10] Kryštof Hoder, Nikolaj Bjørner, and Leonardo Mendonça de Moura. μZ- an efficient engine for fixed points with constraints. In *CAV*, pages 457–462, 2011.

[11] Andreas Podelski and Andrey Rybalchenko. ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In *PADL*, pages 245–259, 2007.

[12] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *PLDI*, pages 159–169, 2008.

[13] Geoff Sutcliffe, Stephan Schulz, Koen Claessen, and Peter Baumgartner. The TPTP Typed First-Order Form with Arithmetic. In *LPAR*, pages 406–419, 2012.

[14] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. In *SAS*, pages 298–315, 2011.