Input-Output Model Programs*

Margus Veanes and Nikolaj Bjørner

Microsoft Research, Redmond, WA, USA
{margus,nbjorner}@microsoft.com

Abstract. Model programs are used as high-level behavioral specifications typically representing abstract state machines. For modeling reactive systems, one uses input-output model programs, where the action vocabulary is divided between two conceptual players: the input player and the output player. The players share the action vocabulary and make moves that are labeled by actions according to their respective model programs. Conformance between the two model programs means that the output (input) player only makes output (input) moves that are allowed by the input (output) players model program. In a bounded game, the total number of moves is fixed. Here model programs use a background theory $\mathcal{T}$ containing linear arithmetic, sets, and tuples. We formulate the bounded game conformance checking problem, or BGC, as a theorem proving problem modulo $\mathcal{T}$ and analyze its complexity.

1 Introduction

Model programs are typically used to describe protocol-like behavior of software systems, with the underlying update semantics based on abstract state machines or ASMs [17]. At Microsoft, model programs are used for model-based testing of public application-level network protocols in the Windows organization, as an integral part of the protocol quality assurance process [16]. In such models, the action vocabulary is often divided into controllable and observable actions, reflecting the testers point of view, i.e., what actions are controllable by the tester versus what actions are observable by the tester. The central problem is to determine if an implementation *conforms* to a given specification. In the presence of controllable and observable actions, the problem can be described as a game conformance checking problem, where the tester executes controllable actions and the implementation responds with observable actions. Traditionally, model-based conformance testing is a black-box testing technique where the actual implementation code is assumed to be unknown to the tester.

In this paper we look at the game conformance checking problem from the symbolic (or static) analysis point of view. The implementation is not

* Microsoft Research Technical Report MSR-TR-2009-56.

a “black box” but a “gray box”. In other words, the implementation is also assumed to be given as a model program through some abstraction function.

The general game conformance checking problem is very hard but can be approximated in various ways. A natural approximation is to bound the number of steps or moves that the players make. This corresponds directly to the fact that actual tests have a finite length. The problem we introduce and analyze in this paper is the Bounded Game Conformance problem of model programs, or BGC for short. We translate the problem into a theorem proving problem modulo a background theory that is most commonly needed in model programs, and analyze the complexity of the problem. For a class of model programs that are common in practice the problem is shown to be decidable. We also discuss a concrete analysis approach for BGC using a satisfiability modulo theories (SMT) based theorem prover Z3.

What differentiates model programs from traditional sequential programs is that model programs typically assume a rich background universe and often operate on a more abstract level, for example, they use set comprehensions and parallel updates to compute a collection of elements in a single atomic step, rather than one element at a time, in a loop. A model program whose action vocabulary is divided into two disjoint parts (corresponding to two players), is called an *input-output model program*. Figure 1 illustrates two input-output model programs written in AsmL [3, 18]. The *Spec* model program in Figure 1 is an abstracted version of the cancellation feature in the SMB2 protocol [22] that is a successor of the Windows file sharing client-server protocol SMB. The SMB protocol is used for file sharing by Windows machines and machines running third party implementations, such as Samba.

In Section 2 we define model programs formally. In Section 3 we introduce the problem of *bounded game conformance checking* or *BGC* and show its reduction to a theorem proving problem modulo $\mathcal{T}$. Section 4 discusses the complexity of *BGC*. Section 5 discusses implementation of *BGC* using Z3 [13]. Section 6 is about related work.

2 Model programs

We consider a background $\mathcal{T}$ that includes linear arithmetic, Booleans, tuples, and sets. All values in $\mathcal{T}$ have a given *sort*. Well-formed expressions of $\mathcal{T}$ are shown in Figure 2. Each sort corresponds to a disjoint part of the universe. We do not add explicit sort annotations to symbols or

Model program *Spec*

```
enum Mode
  Undef    = 0
  Sent     = 1
  Canceled = 2
var M as Map of Integer to Mode
  = {->}

[i,Action]
Req(m as Integer)
  require m notin M
  M(m) := Sent

[i,Action]
Cancel(m as Integer)
  require true
  if M(m) = Sent
    M(m) := Canceled

[o,Action]
Res(m as Integer, b as Boolean)
  require m in M and
          (b or M(m) = Canceled)
  remove m from M
```

Model program *Impl*

```
var R as Set of Integer = {}

[i,Action]
Req(m as Integer)
  require true
  add m to R

[i,Action]
Cancel(m as Integer)
  require true
  skip

[o,Action]
Res(m as Integer, b as Boolean)
  require (m in R) and b
  remove m from R
```

Fig. 1. Here *Req* and *Cancel* are **i**-actions and *Res* is an **o**-action. The model program *Spec* specifies a request cancellation protocol. A request, identified by a message id m, can be Canceled at any time. A response must be associated to some pending request, where if b is false then the request must have been Canceled. The model program *Impl* describes a particular implementation that never cancels any requests, and responds to all requests in some arbitrary order.

expressions but always assume that all expression are well-sorted. A value is *basic* if it is either a Boolean, an integer, or a tuple of basic values.

The expression $\mathit{Ite}(\varphi, t_1, t_2)$ equals t_1 if φ is true, and it equals t_2, otherwise. For each sort, there is a specific *Default* value in the background. In particular, for Booleans the value is *false*, for set sorts the value is $\emptyset$, for integers the value is 0 and for tuples the value is the tuple of defaults of the respective tuple elements.

The function *TheElementOf* maps every singleton set to the element in that set and maps every other set to *Default*. Note that *extensionality* of sets: $\forall v\, w\, (\forall y(y \in v \leftrightarrow y \in w) \rightarrow v = w)$, allows us to use set comprehensions as terms: the *comprehension term* $\{t(\bar{x}) \mid_{\bar{x}} \varphi(\bar{x})\}$ represents the set such that $\forall y(y \in \{t(\bar{x}) \mid_{\bar{x}} \varphi(\bar{x})\} \leftrightarrow \exists \bar{x}(t(\bar{x}) = y \wedge \varphi(\bar{x})))$.

$$
\begin{array}{ll}
T^{\sigma} & ::= x^{\sigma} \mid \mathit{Default}^{\sigma} \mid \mathit{Ite}(T^{\mathbb{B}}, T^{\sigma}, T^{\sigma}) \mid \mathit{TheElementOf}(T^{\mathbb{S}(\sigma)}) \mid \\
 & \quad \pi_i(T^{\sigma_0 \times \cdots \times \sigma_{i-1} \times \sigma \times \cdots \times \sigma_k}) \\
T^{\sigma_0 \times \sigma_1 \times \cdots \times \sigma_k} & ::= \langle T^{\sigma_0}, T^{\sigma_1}, \ldots, T^{\sigma_k} \rangle \\
T^{\mathbb{Z}} & ::= k \mid T^{\mathbb{Z}} + T^{\mathbb{Z}} \mid k * T^{\mathbb{Z}} \\
T^{\mathbb{B}} & ::= \mathit{true} \mid \mathit{false} \mid \neg T^{\mathbb{B}} \mid T^{\mathbb{B}} \wedge T^{\mathbb{B}} \mid T^{\mathbb{B}} \vee T^{\mathbb{B}} \mid T^{\mathbb{B}} \Rightarrow T^{\mathbb{B}} \mid \forall x\, T^{\mathbb{B}} \mid \exists x\, T^{\mathbb{B}} \mid \\
 & \quad T^{\sigma} = T^{\sigma} \mid T^{\mathbb{S}(\sigma)} \subseteq T^{\mathbb{S}(\sigma)} \mid T^{\sigma} \in T^{\mathbb{S}(\sigma)} \mid T^{\mathbb{Z}} \leq T^{\mathbb{Z}} \\
T^{\mathbb{S}(\sigma)} & ::= \{T^{\sigma} \mid_{\bar{x}} T^{\mathbb{B}}\} \mid \emptyset^{\mathbb{S}(\sigma)} \mid T^{\mathbb{S}(\sigma)} \cup T^{\mathbb{S}(\sigma)} \mid T^{\mathbb{S}(\sigma)} \cap T^{\mathbb{S}(\sigma)} \mid T^{\mathbb{S}(\sigma)} \setminus T^{\mathbb{S}(\sigma)} \\
T^{\mathbb{A}} & ::= f^{(\sigma_0, \ldots, \sigma_{n-1})}(T^{\sigma_0}, \ldots, T^{\sigma_{n-1}})
\end{array}
$$

Fig. 2. Well-formed expressions in $\mathcal{T}$. Sorts are shown explicitly here. An expression of sort σ is written T^{σ}. The sorts $\mathbb{Z}$ and $\mathbb{B}$ are for integers and Booleans, respectively, k stands for any integer constant, x^{σ} is a variable of sort σ. The sorts $\mathbb{Z}$ and $\mathbb{B}$ are *basic*, so is the *tuple sort* $\sigma_0 \times \cdots \times \sigma_k$, provided that each σ_i is basic. The *set sort* $\mathbb{S}(\sigma)$ is not basic and requires σ to be basic. All quantified variables are required to have basic sorts. The sort $\mathbb{A}$ is called the *action sort*, $f^{(\sigma_0,\ldots,\sigma_{n-1})}$ stands for an *action symbol* with fixed arity n and argument sorts $\sigma_0, \ldots, \sigma_{n-1}$, where each argument sort is a set sort or a basic sort. The sort $\mathbb{A}$ is *not* basic. The only atomic relation that can be used for $T^{\mathbb{A}}$ is equality. $\mathit{Default}^{\mathbb{A}}$ is a nullary action symbol. Boolean expressions are also called *formulas* in the context of $\mathcal{T}$. In the paper, sort annotations are mostly omitted but are always assumed.

Actions. There is a specific *action sort* $\mathbb{A}$, values of this sort are called *actions* and have the form $f(v_0, \ldots, v_{\mathrm{arity}(f)-1})$. $\mathit{Default}^{\mathbb{A}}$ has arity 0. Two actions are equal if and only if they have the same action symbol and their corresponding arguments are equal. An action $f(\bar{v})$ is called an *f-action*. Every action symbol f with arity $n > 0$, is associated with a unique *parameter variable* f_i for all i, $0 \leq i < n$.[1]

An *assignment* is a pair $x := t$ where x is a variable and t is a term (both having the same sort). An *update rule* is a finite set of assignments where the assigned variables are distinct. In the following definition, internal non-determinism of model programs (through choice variables [7]) is excluded, the initial state condition is omitted, and all state variables must be updated by each action. The last two restrictions are without loss of generality, and allow us to provide a simplified view of the definitions.

Definition 1 (Input-Output Model Program). An *input-output model program* is a tuple $P = (\Sigma, \Gamma^{\mathbf{i}}, \Gamma^{\mathbf{o}}, R)$, where

– Σ is a finite set of variables called *state variables*;

[1] In AsmL one can of course use any formal parameter name, such as m in Figure 1, following standard conventions for method signatures.

- $\Gamma^{\mathbf{i}}$ is a finite set of $\mathbf{i}$-*action symbols*, $\Gamma^{\mathbf{o}}$ is a finite set of $\mathbf{o}$-*action symbols*, $\Gamma^{\mathbf{i}} \cap \Gamma^{\mathbf{o}} = \emptyset$;
- R is a collection $\{R_f\}_{f \in \Gamma^{\mathbf{i}} \cup \Gamma^{\mathbf{o}}}$ of *action rules* $R_f = (\gamma, U)$, where
  - γ is a formula called the *guard of* f;
  - U is an update rule $\{x := t_x\}_{x \in \Sigma}$, called the *update rule of* f.

  All free variables in R_f must be in $\Sigma \cup \{f_i\}_{i < \mathrm{arity}(f)}$.

We often say *action* to also mean an action rule or an action symbol, if the intent is clear from the context. In the following, we say model program for input-output model program. The following special class of model programs is important when considering analysis.

Definition 2 (Basic Model Programs). A model program is *basic* if all parameter variables in it are basic.

Standard *ASM update rules* can be translated into update rules of model programs. A detailed translation from standard ASMs to model programs is given in [7]. In the general case, model programs also use *maps*, e.g., M is a map in *Spec* in Figure 1, that are used to represent dynamic functions of ASMs. In $\mathcal{T}$, maps are represented by their graphs as sets of pairs, see [7].

States. A *state* is a mapping of variables to values. Given a state S and an expression E, where S maps all the free variables in E to values, E^S is the *evaluation of* E *in* S. Given a state S and a formula φ, $S \models \varphi$ means that φ is true in S. A formula φ is *valid* (in $\mathcal{T}$) if φ is true in all states. *Since* $\mathcal{T}$ *is assumed to be the background theory we usually omit it, and assume that each state also has an implicit part that satisfies* $\mathcal{T}$. In the following let $P = (\Sigma, \Gamma^{\mathbf{i}}, \Gamma^{\mathbf{o}}, R)$ be a fixed model program.

Definition 3. An action $a = f(v_0, \ldots, v_{n-1})$ is *enabled* in a state S if $S' = S \cup \{f_i \mapsto v_i\}_{i<n}$ satisfies the guard of f. If a is enabled in S then a *causes a transition* from S to the state $S_1 = \{x \mapsto t_x^{S'}\}_{x \in \Sigma}$, denoted by $S \xrightarrow{a} S_1$.

An *input-output labeled transition system* or *LTS* for short is a tuple $(\mathcal{S}, S^0, L^{\mathbf{i}}, L^{\mathbf{o}}, T)$, where $\mathcal{S}$ is a set of *states*, $S^0 \in \mathcal{S}$ is an *initial state*, $L = L^{\mathbf{i}} \cup L^{\mathbf{o}}$ is a set of *labels*, where $L^{\mathbf{i}} \cap L^{\mathbf{o}} = \emptyset$, and $T \subseteq \mathcal{S} \times L \times \mathcal{S}$ is a *transition relation.*

Definition 4. $[\![P]\!]$ is the LTS $(\mathcal{S}, S^0, L^{\mathbf{i}}, L^{\mathbf{o}}, T)$; $S^0 = \{x \mapsto \mathit{Default}\}_{x \in \Sigma}$; $L^{\mathbf{i}}$ ($L^{\mathbf{o}}$) is the set of all actions over $\Gamma^{\mathbf{i}}$ ($\Gamma^{\mathbf{o}}$); T and $\mathcal{S}$ are the least sets such that, $S^0 \in \mathcal{S}$, and if $S \in \mathcal{S}$ and $S \xrightarrow{a} S_1$ then $(S, a, S_1) \in T$.

Given an action sequence $\alpha = (a_0, \ldots, a_{k-1})$ and transitions $S_i \xrightarrow{a_i} S_{i+1}$ for $0 \le i < k$, of an LTS, we write $S_0 \xrightarrow{\alpha} S_k$. If S_0 is the initial state then α is called a *trace* of the LTS. The set of all traces of P is denoted by $\mathit{Traces}(P)$.

3 Bounded Game Conformance

The basic notion of conformance between two (input-output) model programs is based on the notion of *alternating simulation* between two LTSs. Definition 5 below is consistent with [11], and is based on [2]. The definition makes the assumption that the LTSs are *deterministic*, i.e., for any two transitions $S \xrightarrow{a} S'$ and $S \xrightarrow{a} S''$, $S' = S''$. Thus, LTSs are viewed here as *interface automata* [12] and the transition relation becomes a transition function. Note that $[\![P]\!]$ is deterministic for a model program P.[2] Let $M_i = (\mathcal{S}_i, S_i^0, L^{\mathbf{i}}, L^{\mathbf{o}}, T_i)$, for $i = 1, 2$, be deterministic LTSs.

Definition 5 ($\preceq$). $M_1 \preceq M_2$ iff there exists an alternating simulation ρ from M_1 to M_2 such that $(S_1^0, S_2^0) \in \rho$, where an *alternating simulation from M_1 to M_2* is a relation $\rho \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ such that, for all $(S_1, S_2) \in \rho$:

- For all $a \in L^{\mathbf{o}}$, if $S_1 \xrightarrow{a} S_1'$ then $S_2 \xrightarrow{a} S_2'$ and $(S_1', S_2') \in \rho$.
- For all $a \in L^{\mathbf{i}}$, if $S_2 \xrightarrow{a} S_2'$ then $S_1 \xrightarrow{a} S_1'$ and $(S_1', S_2') \in \rho$.

Example 1. Consider the following two model programs, where $s^0 := \emptyset$ is the initial and only state, and *in* and *out* are nullary action symbols.

$$\begin{aligned}
\mathit{Spec}_{\mathit{trivial}} &= (\emptyset, \{\mathit{in}\}, \{\mathit{out}\}, \{(\mathit{false}, \emptyset)_{\mathit{in}},\ (\mathit{true}, \emptyset)_{\mathit{out}}\}) \\
\mathit{Impl}_{\mathit{trivial}} &= (\emptyset, \{\mathit{in}\}, \{\mathit{out}\}, \{(\mathit{true}, \emptyset)_{\mathit{in}},\ (\mathit{false}, \emptyset)_{\mathit{out}}\}) \\
[\![\mathit{Spec}_{\mathit{trivial}}]\!] &= (\{s^0\}, s^0, \{\mathit{in}\}, \{\mathit{out}\}, \{(s^0, \mathit{out}, s^0)\}) \\
[\![\mathit{Impl}_{\mathit{trivial}}]\!] &= (\{s^0\}, s^0, \{\mathit{in}\}, \{\mathit{out}\}, \{(s^0, \mathit{in}, s^0)\})
\end{aligned}$$

Clearly $[\![\mathit{Impl}_{\mathit{trivial}}]\!] \preceq [\![\mathit{Spec}_{\mathit{trivial}}]\!]$ and $[\![\mathit{Spec}_{\mathit{trivial}}]\!] \not\preceq [\![\mathit{Impl}_{\mathit{trivial}}]\!]$.

⊠

The following characterization of $\preceq$ in terms of traces, follows from Definition 5 and is used below.

Lemma 1. *$N \not\preceq M$ iff there exists a trace α that is a trace of both N and M, and there is an $\mathbf{o}$-label ($\mathbf{i}$-label) a such that (α, a) is a trace of N (M) but not a trace of M (N).*

[2] This is not the case when choice variables are allowed in model programs.

For symbolic analysis, we are primarily interested in the approximations $\preceq_n$ of $\preceq$ where the depth $n \geq 0$ is bounded.

Definition 6 ($\preceq_n$). $M_1 \preceq_n M_2 \overset{\text{def}}{=} M_1 \preceq_n^{(S_1^0, S_2^0)} M_2$ where $M_1 \preceq_n^{(S_1,S_2)} M_2$ iff, either $n = 0$, or the following holds:

– For all $a \in L^{\mathbf{o}}$, if $S_1 \xrightarrow{a} S_1'$ then $S_2 \xrightarrow{a} S_2'$ and $M_1 \preceq_{n-1}^{(S_1',S_2')} M_2$.
– For all $a \in L^{\mathbf{i}}$, if $S_2 \xrightarrow{a} S_2'$ then $S_1 \xrightarrow{a} S_1'$ and $M_1 \preceq_{n-1}^{(S_1',S_2')} M_2$.

It follows easily from the definitions that $M_1 \preceq M_2$ iff $M_1 \preceq_n M_2$ for all $n \geq 0$.

Let P and Q be fixed model programs with the same action vocabularies.

Definition 7. *Q n-refines P*, $Q \preceq_n P$, iff $[\![Q]\!] \preceq_n [\![P]\!]$.

Intuitively, when P is a specification model program and Q is an implementation model program and $Q \preceq_n P$, then Q behaves as expected by P within n steps. Such bounded refinement (or a generalization of it with object-bindings) is used as the underlying notion of conformance in testing of reactive systems in [27], in particular, it is checked in the context of online testing [28]. The bound is due to the fact that tests are finite.

Example 2. Let *Impl* and *Spec* be as in Figure 1. One can show that *Impl* $\preceq_n$ *Spec* for all n and thus *Impl* $\preceq$ *Spec*. It is also the case that *Spec* $\preceq_1$ *Impl* but *Spec* $\not\preceq_2$ *Impl*; for example the trace $(\mathit{Req}(1), \mathit{Req}(1))$ is a trace of *Impl* but not a trace of *Spec*. ⊠

Definition 8 (BGC). *Bounded Game Conformance* problem or *BGC* is the problem of deciding if $Q \preceq_k P$.

In order to reduce BGC into a theorem proving problem, we construct a special formula from given P, Q and n, as defined in Definition 9. Given an expression E and a step number $i > 0$, we write $E[i]$ below for a copy of E where each (unbound) variable x in E has been uniquely renamed to a variable $x[i]$. We assume also that $E[0]$ is E. The intuition for the notation $P_{\mathbf{i}}$ and $P_{\mathbf{o}}$ below is that $P_{\mathbf{i}}$ is the "owner" of **i**-actions ($P_{\mathbf{i}}$ is the specification), and $P_{\mathbf{o}}$ is the "owner" of **o**-actions ($P_{\mathbf{o}}$ is the implementation).

Definition 9 (BGC Formula). Let $P_{\mathbf{i}}$ and $P_{\mathbf{o}}$ be model programs $(\overline{x_\star}, \Gamma^{\mathbf{i}}, \Gamma^{\mathbf{o}}, (\gamma_{f,\star}, U_{f,\star})_{f\in\Gamma^{\mathbf{i}}\cup\Gamma^{\mathbf{o}}})$, for $\star = P_{\mathbf{i}}, P_{\mathbf{o}}$. Assume that $\overline{x_{P_{\mathbf{i}}}} \cap \overline{x_{P_{\mathbf{o}}}} =$

$\emptyset$.[3] Let $\widehat{\mathbf{i}} = \mathbf{o}$ and $\widehat{\mathbf{o}} = \mathbf{i}$. The *BGC formula* for $P_{\mathbf{i}}$, $P_{\mathbf{o}}$, and n is:

$$\begin{aligned}
BGC(P_{\mathbf{o}}, P_{\mathbf{i}}, n) &\stackrel{\text{def}}{=} (\overline{x_{P_{\mathbf{o}}}} = \overline{\mathit{Default}} \wedge \overline{x_{P_{\mathbf{i}}}} = \overline{\mathit{Default}}) \Rightarrow \mathit{Ref}(0, n) \\
\mathit{Ref}(n, n) &\stackrel{\text{def}}{=} \mathit{true} \\
(i < n)\ \mathit{Ref}(i, n) &\stackrel{\text{def}}{=} \bigwedge_{\mathbf{p} \in \{\mathbf{i},\mathbf{o}\}} \bigwedge_{f \in \Gamma^{\mathbf{p}}} \\
&\qquad \forall\, \overline{f[i]}\, ((\gamma_{f,P_{\mathbf{p}}}[i] \wedge \bigwedge_{x:=t \in U_{f,P_{\mathbf{p}}}} x[i+1] = t[i] \\
&\qquad\quad \wedge \mathit{action}[i] = f(\overline{f[i]})) \\
&\qquad\quad \Rightarrow (\gamma_{f,P_{\widehat{\mathbf{p}}}}[i] \wedge (\bigwedge_{y:=s \in U_{f,P_{\widehat{\mathbf{p}}}}} y[i+1] = s[i] \\
&\qquad\qquad \Rightarrow \mathit{Ref}(i+1, n))))
\end{aligned}$$

where $\overline{f[i]} = f_0[i] \dots f_{\text{arity}(f)-1}[i]$ are the parameter variables of action f for step i.[4] For each step number i, there is an additional variable $\mathit{action}[i]$ of sort $\mathbb{A}$ that records the selected action for step i.

Note that all parameter variables have distinct names in each step. The only connection between the steps happens via the state variables. Note also that the resulting formula is a universal formula, assuming that the guards and the update rules do not involve quantifiers (e.g. in comprehensions), i.e., in prenex form, all the quantifiers for the parameter variables are universal. This implies that the negation of the BGF formula is well suited for non-BGC checking of basic model programs (where the state variables can be eliminated) using satisfiability modulo $\mathcal{T}$. The sole purpose of the action variables is to enable easy extraction of the action sequence as a witness of the refinement violation.

The following theorem allows us to prove n-refinement by proving that the BGC formula is valid in $\mathcal{T}$.

Theorem 1. *$BGC(P_{\mathbf{o}}, P_{\mathbf{i}}, n)$ is valid in $\mathcal{T}$ iff $P_{\mathbf{o}} \preceq_n P_{\mathbf{i}}$.*

Proof (Sketch). The case $k = 0$ is trivial. Assume $k > 0$. For the direction ($\Longrightarrow$) we assume that $P_{\mathbf{o}} \not\preceq_k P_{\mathbf{i}}$ and get a shortest run of length $l \leq n$ where the last action is either a $\mathbf{i}$-action that is enabled in $P_{\mathbf{i}}$ but not in $P_{\mathbf{o}}$ (or an $\mathbf{o}$-action that is enabled in $P_{\mathbf{o}}$ but not in $P_{\mathbf{i}}$). From this run we can construct a state that satisfies $\neg BGC(P_{\mathbf{o}}, P_{\mathbf{i}}, n)$, using the property that if $\neg BGC(P_{\mathbf{o}}, P_{\mathbf{i}}, l)$ is satisfiable then $\neg BGC(P_{\mathbf{o}}, P_{\mathbf{i}}, l)$ is also satisfiable,

[3] Or just rename the state variables.
[4] Note that the parameter variables of f are shared between $P_{\mathbf{i}}$ and $P_{\mathbf{o}}$.

for $l' > l$ (because if $\gamma_{f,P_{\widehat{\mathbf{p}}}}[l]$ is false then so is the conjunct $\gamma_{f,P_{\widehat{\mathbf{p}}}}[l] \wedge \ldots$). The proof of the direction ($\Longleftarrow$) is similar. ⊠

Relation to BMPC. There is an alternative way how $\preceq_n$ can be analyzed: by reducing $\not\preceq_n$ to the BMPC problem [7]. *BMPC* is the problem: given a model program P, a reachability condition φ and step bound k, does there exist a trace α of length at most k such that $S^0_{[\![P]\!]} \xrightarrow{\alpha} S$ and $S \models \varphi$.

For this reduction we use *product* of model programs. Let

$$P_i = (\Sigma_i, \Gamma^{\mathbf{i}}, \Gamma^{\mathbf{o}}, \{(\gamma_{f,i}, U_{f,i})\}_{f \in \Gamma})$$

for $i = 1, 2$, where Σ_1 and Σ_2 are disjoint and $\Gamma = \Gamma^{\mathbf{i}} \cup \Gamma^{\mathbf{o}}$.

$$P_1 \otimes P_2 \stackrel{\text{def}}{=} (\Sigma_1 \cup \Sigma_2, \Gamma^{\mathbf{i}}, \Gamma^{\mathbf{o}}, \{(\gamma_{f,1} \wedge \gamma_{f,2}, U_{f,1} \cup U_{f,2})_{f \in \Gamma}\})$$

The following property holds for the product construction.

Lemma 2. *Traces*$(P_1 \otimes P_2) =$ *Traces*$(P_1) \cap$ *Traces*(P_2).

Define the *game conformance invariant* as the following formula:

$$\mathit{Inv}_{\preceq}(P_1, P_2) \stackrel{\text{def}}{=} \forall (\bigwedge_{f \in \Gamma^{\mathbf{o}}} (\gamma_{f,1} \Rightarrow \gamma_{f,2})) \wedge (\bigwedge_{f \in \Gamma^{\mathbf{i}}} (\gamma_{f,2} \Rightarrow \gamma_{f,1}))$$

The following holds.

Theorem 2. *$P_1 \not\preceq_n P_2$ iff $\neg \mathit{Inv}_{\preceq}(P_1, P_2)$ is reachable in $P_1 \otimes P_2$ within n steps.*

Proof. ($\Longrightarrow$) Assume $P_1 \not\preceq_n P_2$. By Lemma 1 there is a trace α of length m of some $m < n$ such that $\alpha \in$ *Traces*(P_1) and $\alpha \in$ *Traces*(P_2) and there is either an output action a such that (α, a) is in *Traces*(P_1) but not in *Traces*(P_2) or an input action a such that (α, a) is in *Traces*(P_2) but not in *Traces*(P_1). It follows that $\mathit{Inv}_{\preceq}(P_1, P_2)$ must be false in the state reached by α. Moreover $\alpha \in$ *Traces*$(P_1 \otimes P_2)$, by Lemma 2.

($\Longleftarrow$) Similar to ($\Longrightarrow$), by using Lemma 2 and Lemma 1. ⊠

Relation to **ioco**. A common notion of conformance that is used for testing reactive systems is **ioco** [23] that stands for **i**nput-**o**utput **co**nformance. There are also several variations of **ioco**, discussed in [23], that are used for testing various extensions of reactive systems. Here we only look at basic **ioco** and consider traces that exclude *quiescence* δ.

The rationale behind excluding δ as a special action is that, in a model program, δ can be defined as a nullary **o**-action with an empty update rule[5] and a guard that is the negation of the existential closure of the conjunction of the guards of all the other **o**-actions. Thus, δ is enabled in a state S iff no other **o**-action is enabled in S and $S \xrightarrow{\delta} S$. Let P^δ denote a model program where δ is defined in this way.

Example 3. Consider the model program *Impl* in Figure 1, where *Res* is the only **o**-action. In Impl^δ, δ has the guard $\neg\exists m\, b\, (m \in R \wedge b)$, that is equivalent to $R = \emptyset$. Similarly, in Spec^δ, δ has the guard $M = \emptyset$.

An LTS M is *input-enabled* if in all states in M that are reachable from the initial state, all **i**-labels are enabled.[6] For example, *Impl* in Figure 1 is input-enabled. The following definition of **ioco** is consistent with the definition in [23] (provided that δ is defined as above).

Definition 10 (ioco). *Let M and N be LTSs over the same* **i***-labels and* **o***-labels. Assume N is input-enabled. N* **ioco** *M iff, for all traces α of M, if there is an* **o***-label a such that (α, a) is a trace of N then (α, a) is a trace of M.*

The relationship between **ioco** and $\preceq$ has been somewhat unclear in the testing community (see for example the discussion in [28]). In our context, $\preceq$ is a generalization of **ioco**. The particular advantage of using $\preceq$ instead of **ioco** is that $\preceq$ is compositional. The definition of $\preceq$ can also be generalized to non-deterministic LTSs, in such a way that the theorem holds when P and Q include choice variables.

Theorem 3. *If $[\![Q]\!]$ is input-enabled then $[\![Q]\!]$* **ioco** *$[\![P]\!] \iff Q \preceq P$.*

Proof. By Lemma 1 and the assumption that $[\![Q]\!]$ is input-enabled. The assumption is needed for the direction $\Longrightarrow$. ⊠

For the *bounded* version of **ioco** we restrict the length of the traces by a given bound n so that all traces in Definition 10 have a length that is at most n; denoted here by $\mathbf{ioco}_n$. We get the following corollary of Theorem 1 and Theorem 3.

Corollary 1. *If $[\![Q]\!]$ is* **i***-enabled then, $BGC(Q, P, n)$ is valid in $\mathcal{T}$ iff $[\![Q]\!]$ $\mathbf{ioco}_n$ $[\![P]\!]$.*

[5] An empty update rule is equivalent to the trivial update rule $\{x := x\}_{x \in \Sigma}$.
[6] Such LTSs are called *input-output transition systems* in [23].

4 Complexity of BGC

The general BGC problem over arbitrary model programs is highly undecidable. This follows from the well-known result that the validity problem of formulas in Presburger arithmetic with unary relations is Π^1_1-complete [1, 19]. Using this result, it is enough to consider model programs that have one action with a single set-valued parameter and a linear arithmetic formula as the guard. To show inclusion in Π^1_1, one can use the same argument that is used in [7] to show that the BMPC problem is in Σ^1_1.

Corollary 2. *BGC is Π^1_1-complete.*

Even when all sets in the background are required to be finite the validity problem in $\mathcal{T}$ over finite sets is still co-re-complete [7].

Corollary 3. *BGC over finite sets is co-re-complete.*

Even though the general BGC problem is undecidable, we are primarily concerned about practical applications. In most model programs, such as the ones in Figure 1, that are used to specify protocols (see also [21, 30]), the actions typically only use *basic* parameters, i.e., parameters whose sort is not a set sort. In other words, our main target for analysis are *basic* model programs (recall Definition 2).

Theorem 4. *BGC of basic model programs is decidable. Moreover, the upper bound of the computational complexity is $2^{2^{2^{cn}}}$ and the lower bound is $2^{2^{cn}}$, where c is a constant and n is the size of the input (P, Q, k).*

Proof (Sketch). Consider the formula $\psi = BGC(Q, P, k)$. First, the formula ψ is translated into logic without sets but with unary relations, by replacing set variables with unary relations and by eliminating set comprehensions and set operations in the usual way, e.g., $t \in S$, where S is a set variable, becomes the atom $R_S(t)$, where R_S is a unary relation symbol. Let the resulting formula be φ. Next, introduce auxiliary predicates that define all the subformulas of φ, by applying the Tseitin transformation [24] to φ. Subsequently, eliminate those auxiliary predicates (as a form of de-Skolemization), by introducing additional quantifiers. (A similar elimination technique that can be used here follows from [15, p 129], see also [25]). The overall reduction implies that the computational complexity of BGC of basic model programs, regarding both the lower and the upper bound, is the same as that of Presburger arithmetic [15]. $\boxtimes$

A naïve implementation of definition 9 could repeat the recursive calls to an exponential number of times. However, note that the results are all shared common sub-expressions.

From a practical perspective, the actual computational complexity of BGC over basic model programs problem depends on the quantifier alternation depth. In many problems the final formula is universal, because quantifiers are not used inside guards or update rules.

5 Implementation

We created a prototype for testing the Bounded Game Conformance formulas generated from definition 9[7]. The prototype uses the F# programmatic interface to the state-of-the art SMT solver Z3 [13] to represent Input-Output Model Programs as a collection of transition pairs. Each pair consists of a specification and an implementation transition and is tagged as either **i** or **o** to indicate which direction to check the alternating simulation. The data-types used in the model program are mapped directly to native Z3 theories. For example, the `Mode` enumeration type is mapped into a special case of algebraic data-types where enumerations are encoded as nullary constructors.

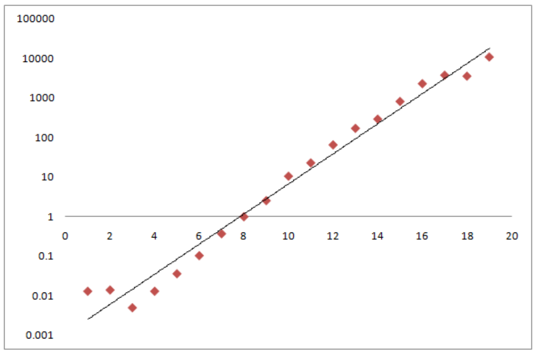

Fig. 3. Timing $BGC(P_{\mathbf{o}}, P_{\mathbf{i}}, n)$, $n = 1..19$.

The finite map M is represented as an array, and the theory of extensional arrays is used to handle the operations on M. Similarly, the set R is represented as an array that maps integers to Booleans. The operations, element-wise addition and removal required by Req and Res) are simply array updates. Z3 supports richer set operations as an extension to the theory of arrays, but this example does not make use of these. The prototype uses the fact that terms in Z3 are internally represented as shared directed acyclic graphs. In particular, the repeated occurrences of $Ref(i+1, n)$ represent the same formula. The formula is built only once, and reused in the different occur-

[7] See http://research.microsoft.com/en-us/people/nbjorner/ictac09.zip.

rences. The size of the resulting path formula is therefore proportional to the number of unfoldings n and to the size of the input model program.

On the other hand, the size of the input does not depend on the size of the state space. The potentially unbounded size of the state space is also not a factor when checking for bounded game conformance, but our techniques are sensitive to the number of paths in the unfoldings. Figure 3 shows the number of seconds it took Z3 to check for conformance for up to 19 unfoldings for our example in Figure 1. We observe that the time overhead grows exponentially with the number of unfoldings n (so linear in the number of paths that are checked). Not shown is the space overhead, which was very modest: space consumption during solving grew linearly with n, from 12 MB to around 20 MB. Figure 4 shows the similar timings required for checking the equivalent property $Inv_{\preceq}(P_1, P_2)$ for the BMPC formulation. The overhead of checking the invariant in this formulation is still exponential, but the growth is much slower and it is therefore possible to explore up to 47 unfoldings, with each check taking less than 20 minutes.

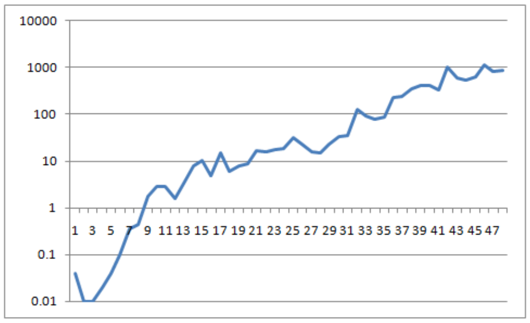

Fig. 4. Timing $Inv_{\preceq}(P_1, P_2)$, $n = 1..47$.

A more interesting use of bounded conformance checking is to detect bugs in the models used for either the specifications or implementations. We can *plant* a bug in our example from Figure 1 by changing the *Impl* transition *Res* to forget removing m from R. The bogus transition is therefore:

```
[o,Action]
Res(m as Integer, b as Boolean)
  require (m in R) and b
  skip
```

It takes Z3 well below a second to create a counter-example of length 3. Since the $BGC(P_{\mathbf{o}}, P_{\mathbf{i}}, n)$ formula contains equalities that track which actions are taken together with their parameters, it is easy to use Z3's model-producing facilities to extract the counter-example:

```
actions0 -> (req 1)
```

```
actions1 -> (res 1 true)
actions2 -> (res 1 true)
```

The counter-example says that the client request (*Req*) action is applied with input 1, followed by two server responses (*Res*) using the same parameter 1. The *Spec* model program is not enabled in response to this second action.

6 Related work

BGC is related to the bounded model program checking problem or BMPC [7, 26, 29], that is a bounded path exploration problem of a given model program. BMPC is a generalization of bounded model checking to model programs. The technique of bounded model checking by using SAT solving was introduced in [4] and the extension to SMT was introduced in [14]. BMPC reduces to satisfiability modulo $\mathcal{T}$. BMPC can be reduced in polynomial time to BGC, providing the computational complexity bounds for BMPC, using Theorem 4, that are left open in [7]. Unlike BGC, the BCC [25] problem introduces k-depth quantifier alternation in the resulting formula, where k is the step bound. This is also the case for a generalization of BGC for non-deterministic model programs, in which case the reduction to BMPC, shown in Section 3, does not work. The resulting formula for a BMPC problem does not have quantifier alternation, even for non-deterministic model programs, since choice variables and parameter variables are treated equally.

Symbolic analysis of refinement relations through theorem proving are used in hardware [10, 9]. Various refinement problems between specifications are also the topic of many analysis tools, where sets and maps are used as foundational data structures, such as ASMs, RAISE, Z, TLA+, B, see [5], where the techniques introduced here could be applied. In some cases, like in RAISE, the underlying logic is three-valued. In many of the formalisms, frame conditions need to be specified explicitly, and are not implicit as in the case of model programs or ASMs. In Alloy [20], the analysis is reduced to SAT, by finitizing the data types. In our case we bound the search depth rather than the size of the data types

For implementation, we use the state of the art SMT solver Z3 [13], discussed in Section 5. Implementation of the reduction of BGC of basic input-output model programs to linear arithmetic, based on Theorem 4, is future work. In that context the reduction to Z3 can take advantage of built-in support for *Ite* terms, sets, algebraic data-types, and tuples.

The background theory $\mathcal{T}$ can also be extended to include reals, that are natively supported in Z3. Our experiment indicated that Z3 could be used for modest bounded exploration. More interestingly, it posed an intriguing challenge for solvers like Z3 to better handle *diamond* structured formulas. One technique for handling *diamond* style formulas is explored in [6]. It uses a combination of abstract interpretation and constraint propagation to speed up the underlying constraint solving engine.

We see conformance from a game point of view, that view is inspired by [11]. The game view can also be used to formulate other problems related to input-output model programs, such as finding winning strategies to reach certain goal states. In the context of testing, a overview of using games is given in [31]. Game based testing approaches with finite model programs are also discussed in [8] using reachability games.

References

1. R. Alur and T. A. Henzinger. A really temporal logic. In *Proc. 30th Symp. on Foundations of Computer Science*, pages 164–169, 1989.
2. R. Alur, T. A. Henzinger, O. Kupferman, and M. Vardi. Alternating refinement relations. In *Proceedings of the Ninth International Conference on Concurrency Theory (CONCUR'98)*, volume 1466 of *LNCS*, pages 163–178. Springer, 1998.
3. AsmL. http://research.microsoft.com/fse/AsmL/.
4. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems, (TACAS'99)*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.
5. D. Bjørner and M. Henson, editors. *Logics of Specification Languages*. Springer, 2008.
6. N. Bjørner, B. Dutertre, and L. de Moura. Accelerating Lemma Learning using Joins - DPPL(Join). In *Proceedings of short papers at LPAR'08*, 2008.
7. N. Bjørner, Y. Gurevich, W. Schulte, and M. Veanes. Symbolic bounded model checking of abstract state machines. Technical Report MSR-TR-2009-14, Microsoft Research, February 2009. Submitted to IJSI.
8. A. Blass, Y. Gurevich, L. Nachmanson, and M. Veanes. Play to test. Technical Report MSR-TR-2005-04, Microsoft Research, January 2005. Short version appears in FATES 2005.
9. R. E. Bryant, S. M. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 470–482. Springer, 1999.
10. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In D. L. Dill, editor, *Conference on Computer-Aided Verification (CAV'94)*, volume 818 of *LNCS*, pages 68–80. Springer, 1994.
11. L. de Alfaro. Game models for open systems. In N. Dershowitz, editor, *Verification: Theory and Practice: Essays Dedicated to Zohar Manna on the Occasion of his 64th Birthday*, volume 2772 of *LNCS*, pages 269–289. Springer, 2004.

12. L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference and the 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 109–120. ACM, 2001.
13. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, (TACAS'08)*, LNCS. Springer, 2008.
14. L. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *Proceedings of the 18th International Conference on Automated Deduction (CADE'02)*, volume 2392 of *LNCS*, pages 438–455. Springer, 2002.
15. M. J. Fisher and M. O. Rabin. Super-exponential complexity of presburger arithmetic. In B. F. Caviness and J. R. Johnson, editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 122–135. Springer, 1998. Reprint from *SIAM-AMS Proceedings*, Vol VII, 1974, pp. 27-41.
16. W. Grieskamp, D. MacDonald, N. Kicillof, A. Nandan, K. Stobie, and F. Wurden. Model-based quality assurance of Windows protocol documentation. In *First International Conference on Software Testing, Verification and Validation, ICST*, Lillehammer, Norway, April 2008.
17. Y. Gurevich. *Specification and Validation Methods*, chapter Evolving Algebras 1993: Lipari Guide, pages 9–36. Oxford University Press, 1995.
18. Y. Gurevich, B. Rossman, and W. Schulte. Semantic essence of AsmL. *Theor. Comput. Sci.*, 343(3):370–412, 2005.
19. J. Y. Halpern. Presburger arithmetic with unary predicates is Π^1_1 complete. *Journal of Symbolic Logic*, 56:637–642, 1991.
20. D. Jackson. *Software Abstractions*. MIT Press, 2006.
21. J. Jacky, M. Veanes, C. Campbell, and W. Schulte. *Model-based Software Testing and Analysis with C#*. Cambridge University Press, 2008.
22. SMB2. http://msdn2.microsoft.com/en-us/library/cc246482.aspx, 2008.
23. J. Tretmans. Model based testing with labelled transition systems. In R. Hierons, J. Bowen, and M. Harman, editors, *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 1–38. Springer, 2008.
24. G. S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Mathematics and Mathematical Logic*, Part II:115–125, 1968.
25. M. Veanes and N. Bjørner. Symbolic bounded conformance checking of model programs. Technical Report MSR-TR-2009-28, Microsoft Research, March 2009.
26. M. Veanes, N. Bjørner, and A. Raschke. An SMT approach to bounded reachability analysis of model programs. In *FORTE'08*, volume 5048 of *LNCS*, pages 53–68. Springer, 2008.
27. M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Model-based testing of object-oriented reactive systems with Spec Explorer. In R. Hierons, J. Bowen, and M. Harman, editors, *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 39–76. Springer, 2008.
28. M. Veanes, C. Campbell, W. Schulte, and N. Tillmann. Online testing with model programs. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 273–282. ACM, 2005.
29. M. Veanes and A. Saabas. On bounded reachability of programs with set comprehensions. In *LPAR'08*, volume 5330 of *LNAI*, pages 305–317. Springer, 2008.
30. M. Veanes, A. Saabas, and N. Bjørner. Bounded reachability of model programs. Technical Report MSR-TR-2008-81, Microsoft Research, May 2008.

31. M. Yannakakis. Testing, optimization, and games. In *Proceedings of the Nineteenth Annual IEEE Symposium on Logic In Computer Science, LICS 2004*, pages 78–88. IEEE, 2004.