Analysis of Recursive Game Graphs Using Data Flow Equations

K. Etessami

LFCS, School of Informatics
U. of Edinburgh
kousha@inf.ed.ac.uk

Abstract. Given a finite-state abstraction of a sequential program with potentially recursive procedures and input from the environment, we wish to check statically whether there are input sequences that can drive the system into "bad/good" executions. Pushdown games have been used in recent years for such analyses and there is by now a very rich literature on the subject. (See, e.g., [BS92,Tho95,Wal96,BEM97,Cac02a,CDT02].) In this paper we use recursive game graphs to model such interprocedural control flow in an open system. These models are intimately related to pushdown systems and pushdown games , but more directly capture the control flow graphs of recursive programs ([AEY01,BGR01,ATM03b]). We describe alternative algorithms for the well-studied problems of determining both reachability and Büchi winning strategies in such games. Our algorithms are based on solutions to second-order data flow equations, generalizing the Datalog rules used in [AEY01] for analysis of recursive state machines. This offers what we feel is a conceptually simpler view of these well-studied problems and provides another example of the close links between the techniques used in program analysis and those of model checking.

There are also some technical advantages to the equational approach. Like the approach of Cachat [Cac02a], our solution avoids the necessarily exponential-space blow-up incurred by Walukiewicz's algorithms for pushdown games. However, unlike [Cac02a], our approach does not rely on a representation of the space of winning configurations of a pushdown graph by (alternating) automata. Only "minimal" sets of exits that can be "forced" need to be maintained, and this provides the potential for greater space efficiency. In a sense, our algorithms can be viewed as an "automaton-free" version of the algorithms of [Cac02a].

1 Introduction

There has been intense activity in recent years aimed at extending the scope of model checking to finite-state abstractions of sequential programs described by modular and potentially recursive procedures. A partial list of references includes [BS92,Wal96,BEM97,Rep98,EHRS00,BR00,AEY01,BGR01,Cac02a,CDT02]. Pushdown systems are one of the primary vehicles for such analyses. When such models are analyzed in the setting of an open system, where the environment

B. Steffen and G. Levi (Eds.): VMCAI 2004, LNCS 2937, pp. 282–296, 2004.

is viewed as an adversary, a natural model to study becomes pushdown games. There is by now a very rich literature on the analysis of pushdown games (see, e.g., [Cau90,BS92,Tho95,Wal96,BEM97,Cac02a,CDT02].)

In this paper we use recursive game graphs to model such interprocedural control flow in an open system. These models are intimately related to pushdown systems and pushdown games, but more directly capture the control flow graphs of recursive programs. Recursive state machines (RSMs) were introduced and studied in [AEY01] and independently in [BGR01], and are related to similar models studied program analysis (see, e.g., [Rep98]). Besides giving algorithms for their analysis, they showed that RSMs are expressively equivalent to pushdown systems, with efficient translations in both directions. More recently [ATM03b,ATM03a] have studied "modular strategies" on Recursive Game Graphs (RGGs), a natural adaptation of RSMs to a game setting. The translations of [AEY01,BGR01] can easily be adapted to show that pushdown games and (labelled) RGGs are expressively equivalent.

The results of Walukiewicz [Wal96] were a key watershed in the analysis of pushdown games. Besides much else, he showed that determining the existence of winning strategies under any parity encodable winning condition is in EXPTIME, and that existence of winning strategies under simple reachability winning conditions is already EXPTIME-hard. In Walukiewicz's algorithm one first constructs from a pushdown game P an exponentially larger (flat) game graph G_P. A winning strategy in G_P corresponds to a winning strategy in P, and one then solves G_P via efficient algorithms for flat games. The disadvantage of such an algorithm from a practical point of view is that it can not get "lucky": exponential space is consumed in the first phase on any input, even if the game P may be very "simple" to solve. Subsequently, many others have studied algorithms for analysis of pushdown systems and pushdown games. One effective approach has been based on the observation that the reaching configurations of a pushdown system form a regular set ([FWW97,BEM97,EHRS00]). This approach has been used more recently by Cachat and others [Cac02a,CDT02,Cac02b,Ser03] to give alternative algorithms for analysis of pushdown games. These algorithms do not necessarily incur the exponential space blow-up incurred by Walukiewicz's algorithm, but they do require construction of an alternating automaton accepting the winning configurations of a pushdown game.

We describe alternative algorithms for determining both reachability and Büchi strategies on RGGs. Our algorithms do not make use of any automata to represent global configurations of the underlying game graph, but are instead based on solutions to second-order data flow equations over sets of exit nodes in the RGGs. This generalizes the Datalog rules used in [AEY01] for analysis of recursive state machines. It is also closely related to the algorithm in [ATM03b] for computing "modular strategies" for reachability on recursive game graphs. Computing modular strategies is NP-complete, whereas computing arbitrary strategies is EXPTIME-complete, so our algorithms necessarily differ. But our underlying ideas for the reachability algorithm are closely related to theirs, and both can be viewed as generalizations of the approach of [AEY01].

The dataflow equation approach offers what we feel to be a conceptually simpler solution to these well-studied problems. It also provides another example of the close links between the techniques used in program analysis and those of model checking. In a sense, our algorithms can be viewed as an "automaton-free" version of the algorithms of [Cac02a], although they arose in an attempt to generalize the approach of [AEY01].

There are some technical advantages to be gained from using the equational approach. Unlike the automaton approach, in which global winning configurations need to be recorded as accepted strings of an alternating automaton, in our approach only "minimal" sets of exits that can be "forced" need to be maintained and this provides the potential for greater space efficiency when one is interested in, e.g., whether particular vertices can be reached under any "context". Also, a formulation based on solutions of data flow equations allows for the application of well established techniques in program analysis for efficient evaluation, such as worklist data structures to manage sets that require updates (see, e.g., [NNH99,App98]).

The sections of the paper are as follows: in section 2 we provide background definitions on recursive game graphs, in section 3 we provide our algorithm for RGGs under a reachability winning condition, in section 4 we extend these to Büchi conditions, and we conclude in 5.

2 Definitions

Syntax. A *recursive game graph (RGG)* A is given by a tuple $\langle A_1, \ldots, A_k \rangle$, where each *component game graph* $A_i = (N_i \cup B_i, Y_i, En_i, Ex_i, \delta_i)$ consists of the following pieces:

- A set N_i of *nodes* and a (disjoint) set B_i of *boxes*.
- A labelling $Y_i : B_i \mapsto \{1, \ldots, k\}$ that assigns to every box an index of one of the component machines, $A_1, \ldots, A_k$.
- A set of *entry* nodes $En_i \subseteq N_i$, and a set of *exit* nodes $Ex_i \subseteq N_i$.
- A transition relation δ_i, where transitions are of the form (u, v) where:
  1. the source u is either a non-exit node in $N_i \setminus Ex_i$, or a pair (b, x), where b is a box in B_i and x is an exit node in Ex_j, where $j = Y_i(b)$;
  2. the destination v is either a non-entry node in $N_i \setminus En_i$, or a pair (b, e), where b is a box in B_i and e is an entry node in En_j, where $j = Y_i(b)$.

Let $N = \bigcup_{i=1}^{k} N_i$ denote the set of all nodes. For a box b in A_i and an entry e of $Y_i(b)$, we define the pair (b, e) to be a *call.* Likewise, for an exit x of $Y_i(b)$, we say (b, x) is a *return.* We will use the term *vertex* to refer collectively to nodes, calls, and returns that participate in some transition, and we denote this set by $Q = \bigcup_{i=1}^{k} Q_i$. That is, the transition relation δ_i is a set of labelled directed edges on the set Q_i of vertices of A_i. Let $\delta = \cup_i \delta_i$ be the set of all edges in A. In addition to the tuple $\langle A_1, \ldots, A_k \rangle$, we are also given a partition of the vertices Q into two disjoint sets Q^0 and Q^1, corresponding to where it is player 0's and player 1's turn to play, respectively.

Semantics. An RGG defines a global game graph $T_A = (V, V_0, V_1, \Delta)$. The global *states* V of T_A, for RGG A, are tuples $\langle b_1, \ldots, b_r, u\rangle$, where $b_1, \ldots, b_r$ are boxes and u is a vertex (not just a node). The global transition relation Δ is given as follows: let $s = \langle b_1, \ldots, b_r, u\rangle$ be a state with u a vertex in Q_j, and $b_r \in B_m$. Then, $(s, s') \in \Delta$ iff one of the following holds:

1. $(u, u') \in \delta_j$ for a vertex u' of A_j, and $s' = \langle b_1, \ldots, b_r, u'\rangle$.
2. u is a call vertex $u = (b', e)$, of A_j, and $s' = \langle b_1, \ldots, b_r, b', e\rangle$.
3. u is an exit-node of A_j, and $s' = \langle b_1, \ldots, b_{r-1}, (b_r, u)\rangle$.

Case 1 corresponds to when control stays within the component A_j, case 2 is when a new component is entered via a box of A_j, case 3 is when the control exits A_j and returns back to A_m.
The global states are partitioned into sets V_0 and V_1, as follows: $s = \langle b_1, \ldots, b_r, u\rangle$ is in V_0 (V_1) if $u \in Q^0$ ($u \in Q^1$, respectively). We augment RGGs with an acceptance condition $\mathcal{F}$. We restrict ourselves to Büchi conditions $F \subseteq N$. The global game graph T_A, together with a start state *init* $\in N$ and acceptance condition F, define an (infinite) game graph with an acceptance condition, $B_A = (V, V_0, V_1, \Delta, \langle init\rangle, F^\star)$, where $F^* = \{\langle \bar{b}, v\rangle \mid v \in F\}$. B_A defines a game as follows. The game begins at $s_0 = \langle init\rangle$. Thereafter, whenever we are at a state s_i in V_0 (V_1), Player 0 (respectively, Player 1), moves by choosing some transition $(s_i, s_{i+1}) \in \Delta$. A run (or *play*) $\pi = s_0 s_1 \ldots$ is constructed from the infinite sequence of moves. The run π is *accepting* if it is accepted under the Büchi condition $F^\star$, meaning for infinitely many indices i, $s_i \in F^\star$. We say that player 0 wins if π is accepting, and 1 wins if π is not accepting. A player is said to have a *winning strategy* in the Büchi game if it can play in such a way that it will win regardless of how the other player plays (we omit a more formal description of a winning strategy). We will also be interested in simpler reachability winning conditions. Given A, a vertex $u \in Q$, and a set of nodes $Z \subseteq N$, define $u \Rightarrow^0 Z$ to mean that player i has a strategy to reach some state $\langle b_1, \ldots b_r, v\rangle$ such that $v \in Z$, from the state $\langle u\rangle$ in the global game graph T_A. We will be interested in two algorithmic problems:

1. *Winning under reachability conditions:* Given A, a vertex $u \in Q$, and a set of nodes $Z \subseteq N$, determine whether $u \Rightarrow^0 Z$.
2. *Winning under Büchi conditions:* Given A, *init* $\in N$, and $F \subseteq N$, determine whether one player or the other has a winning strategy in the game defined by B_A under Büchi acceptance conditions.

3 Algorithm for the Reachability Game

To check whether $v \Rightarrow^0 Z$, we will incrementally associate to each vertex v of component A_i a set-of-sets $RSet_Z(v) \subseteq 2^{Ex_i}$ of exit nodes from Ex_i. Usually, when Z is clear from the context, we write $RSet(v)$ instead of $RSet_Z(v)$. The empty set $\{\}$ will eventually end up in $RSet(v)$ iff $v \Rightarrow^0 Z$. We first make a more general definition. For u a vertex in component A_i, and $X \subseteq Ex_i$, and for an arbitrary set of nodes $Z \subseteq N$, We write $u \Rightarrow^0 (Z, X)$, to mean that player 0 can "force the play", starting at $\langle u \rangle$, to either reach a global state $\langle x \rangle$ for some $x \in X$, or else to reach a global state $\langle b_1, \ldots, b_k, z \rangle$, such that $z \in Z$. Note that $u \Rightarrow^0 Z$ if and only if $u \Rightarrow (Z, \emptyset)$. As we will see, some subset $Y \subseteq X$ will eventually end up in $RSet(u)$ if and only if $u \Rightarrow^0 (Z, X)$. The algorithm to build $RSet(v)$'s proceeds as follows:

1. **Initialization:** for every $z \in Z$, we initialize $RSet(z) := \{\{\}\}$ to the set containing only the empty set. To each exit point $x \in Ex_i$, not in Z, in any component A_i, we associate the set $RSet(x) := \{\{x\}\}$. To all other vertices v we initially associate $RSet(v) := \{\}$.
2. **Rule application:** Inductively, we make sure the relationships defined by rules (a), (b), and (c), below, hold between the sets associated with vertices, based on a standard fixpoint iteration loop. We say an instance of a rule is *applicable* if the right hand side does not equal the left hand side. While there is an applicable instance of a rule, we apply it. For $\mathcal{S} \subseteq 2^D$, over any base set D, let

$$Min(\mathcal{S}) = \{X \in \mathcal{S} \mid \forall\, X' \in \mathcal{S} \text{ if } X' \subseteq X \text{ then } X' = X\}$$

   In other words, $Min(\mathcal{S})$ contains the *minimal* sets $X \in \mathcal{S}$ such that there is no $X' \in \mathcal{S}$ which is strictly contained in X.

   a) For every 0-vertex $v \in Q^0$ that isn't an exit, isn't a call, and isn't in Z:

$$RSet(v) := Min(\bigcup_{(v,w)\in\delta} RSet(w))$$

   In other words, the set of sets associated with v is the union of the set of sets associated with its successors, but only retaining the minimal sets among these. So if it has two successors w_1 and w_2 with $RSet(w_1) = \{\{x_1\}, \{x_2, x_3\}\}$ and $RSet(w_2) = \{\{x_2\}\}$ then $RSet(v) = \{\{x_1\}, \{x_2\}\}$.

   b) For every 1-vertex $v \in Q^1$, with successors $\{w_1, ..., w_k\}$, where v is not an exit, isn't a call, and $v \notin Z$:

$$RSet(v) := Min(\{(\bigcup_{i\in\{1,\ldots,k\}} X_i) \mid \forall i : X_i \in RSet(w_i)\})$$

   In other words, the set of sets associated with each 1-vertex v will consist of all possible unions of sets of its successors, retaining only minimal sets among these. So, if $RSet(w_1) = \{\{x_1\}, \{x_3\}\}$ and $RSet(w_3) = \{\{x_1, x_2\}, \{x_2, x_3\}\}$, then
   $RSet(v) = Min(\{\{x_1, x_2\}, \{x_1, x_2, x_3\}, \{x_2, x_3\}\}) = \{\{x_1, x_2\}, \{x_2, x_3\}\}$.
   Note that if for some w_j, $RSet(w_j) = \{\}$, then $RSet(v) := \{\}$.

   c) For every call vertex (b, e_i), where e_i is an entry point of A_i, A_i has exit set $Ex_i = \{x_1, ..., x_k\}$, and b is in component A_j, where $Y_j(b) = i$:

$$RSet((b, e_i)) := Min(\{\bigcup_{x\in X} X_{b,x} \mid X \in RSet(e_i)\ \&\ X_{b,x} \in RSet((b, x))\})$$

In other words, the set of sets $RSet((b, e_i)) \subseteq 2^{Ex_j}$ associated with the call (b, e_i) consists of the union, for each set $X \in RSet(e_i) \subseteq 2^{Ex_i}$, of the sets $X_{(b,x)} \in RSet((b, x))$, where $x \in X$, and (b, x) is a return of box b. By convention, if $X = \{\}$, then $\bigcup_{x \in X} X_{b,x} = \{\}$. So empty sets carry over from $RSet(e_i)$ to $RSet((b, e_i))$.

For any two sets-of-sets, $\mathcal{S}, \mathcal{S}' \subseteq 2^D$, over any base set D, we will say that $\mathcal{S}'$ *covers* $\mathcal{S}$ *from below*, and denote this by $\mathcal{S}' \sqsubseteq \mathcal{S}$, iff for every set X in $\mathcal{S}$ there is a set Y in $\mathcal{S}'$ such that $Y \subseteq X$. Note that the empty set $\{\}$ is covered from below by any other set, and that the set $\{\{\}\}$ containing only the empty set covers every set from below. (We omit the lattice-theoretic formulation.)

Theorem 1. *For a vertex u of component A_i, a set $Z \subseteq N$, and a set $X \subseteq Ex_i$:*

1. *Let* $\mathrm{RSet}(u)$ *be the set associated with u at some point during the algorithm, and* $\mathrm{RSet}'(u)$ *be the set associated with u some time later. Then* $\mathrm{RSet}'(u) \sqsubseteq \mathrm{RSet}(u)$.
2. *At any time during the algorithm* $\mathrm{RSet}(u)$ *is minimal, i.e., if $Y \in \mathrm{RSet}(u)$ there is no strict subset $Y' \subset Y$, with $Y' \in \mathrm{RSet}(u)$.*
3. *The algorithm will halt, and* $\mathrm{RSet}(u)$ *will get updated at most $2^{|Ex_i|}$ times.*
4. *(*) Some subset $Y \subseteq X$ will eventually end up in* $\mathrm{RSet}(u)$ *if and only if (**) $u \Rightarrow^0 (Z, X)$.*

In particular, letting $X = \emptyset$ in (4.), $u \Rightarrow^0 Z$ if and only if when the algorithm halts $\{\}$ is in $\mathrm{RSet}(u)$.

Proof. Claim (1.) asserts a monotonicity property of these rules. Namely, suppose for vertex u, $RSet(u)$ depends on $RSet(w_i)$ for immediate "neighbours" $\{w_1, \ldots, w_k\}$ (if u is a call, these can be either return vertices of the box, or entries of the corresponding component). If we are about to update $RSet(u)$ using the $RSet'$ sets associated with its neighbours, suppose that for some neighbours, $RSet'(w_i) \sqsubseteq RSet(w_i)$, while for others (that haven't been updated) $RSet'(w_i) = RSet(w_i)$. We can show by inspection that applying each rule results in a set $RSet'(u)$ such that $RSet'(u) \sqsubseteq RSet(u)$.
Claim (2.) follows from both the initial settings of $RSet(u)$'s, and the fact that each of our update rules retains only minimal sets.
Claim (3.) follows, because by claims (1.) and (2.) successive sets associated with a vertex will always cover prior ones from below. Hence, since $\sqsubseteq$ defines a partial-order on these sets, there are no non-trivial chains that repeat the same set. Updates are only performed on applicable rule instances, i.e., when the respective set changes. Hence at most $2^{|Ex_i|}$ updates are performed on $RSet(u)$.
Claim (4.): the easier direction is $(*) \to (**)$: Suppose $Y \subseteq X$, and $Y \in RSet(u)$. By the "soundness" of our rules, there is a way for player 0 to force the game, starting at $\langle u \rangle$, either into a state $\langle y \rangle$ for $y \in Y$, or else into $\langle \bar{b}, z \rangle$, for some $z \in Z$. "Soundness" means: both the initial setting of $RSet(u)$'s, and every rule application preserve the following invariant: if $Y \in RSet(u)$, then $u \Rightarrow^0 (Z, Y)$.
$(**) \to (*)$: Suppose $u \Rightarrow^0 (Z, X)$. Consider Player 0's winning strategy as a finite tree T_{win} with root $\langle u \rangle$, leaves labelled by two kinds of states, either of the

form $\langle x\rangle$, for $x \in X$, or of the form $\langle b_1,\ldots,b_r,z\rangle$, such that $z \in Z$. An internal 1-state of T_{win} has as its children ALL its neighbours in T_A, while an internal 0-state has a single child, one of its neighbours in T_A. In addition, no non-leaf (internal) node is of the form $\langle b_1,\ldots,b_r,z\rangle$, where $z \in Z$.

For $s = \langle b_1,\ldots,b_r,u\rangle$, let $vertex(s) = u$. Extending the notation, for a set of states S let, $vertex(S) = \{vertex(s) \mid s \in S\}$. Consider a particular occurrence of $s = \langle b_1,\ldots,b_r,u\rangle$ in T_{win}, where $u \in Q_i$.[1] We inductively define a "cut off subtree" T^s_{win} of T_{win}, whose root is (this) s and such that for every state s' of T^s_{win}, if $b_1,\ldots,b_r$ is a prefix of every child of s' (and $vertex(s') \notin Z$) then every child of s' in T_{win} is also a child of s' in T^s_{win}. (Note: either every child of s' has $b_1,\ldots,b_r$ as a prefix, or none does because in that case $s' = \langle b_1,\ldots,b_r,ex\rangle$ and $ex \in Ex_i$). Note that when $s = \langle u\rangle$ is the root of T_{win}, $T^s_{win} = T_{win}$. For a finite strategy tree T, let $leafexits(T) = \{s \mid s$ is a leaf of T, and $vertex(s) \notin Z\}$. We will show, by induction on the depth of T^s_{win}, that for every state s in T_{win}, $RSet(vertex(s))$ will eventually contain a set $Y \subseteq vertex(leafexits(T^s_{win}))$. Applying this to the root $\langle u\rangle$ of T_{win}, yields that $(**) \to (*)$.
Base case: if depth of T^s_{win} is 0, then the only state of T^s_{win} is $s = \langle b_1,...,b_r,u\rangle$, and either $u \in Z$, in which case $RSet(u) := \{\{\}\}$, or else u is an exit node, in which case $RSet(u) := \{\{u\}\}$. In either case $RSet(vertex(s))$ will contain $Y = vertex(leafexits(T^s_{win}))$. Inductively: suppose the depth of T^s_{win} is n, with root $\langle b_1,\ldots,b_r,u\rangle$. Let the children of the root be $s_1,\ldots,s_m$. Each child s_i is itself the root of a cut-off subtree $T^{s_i}_{win}$ of depth $\leq n-1$ for player 0, and thus by induction for each i, $RSet(vertex(s_i))$ will eventually contain a set $Y_i \subseteq vertex(leafexits(T^{s_i}_{win}))$. We show that $RSet(u)$ will "after one update" contain a $Y \subseteq vertex(leafexits(T^s_{win}))$. There are several cases based on u:

1. u is an exit node. In this case T^s_{win} will always be a trivial 1 node tree. The set $RSet(u)$ will always be the same as its initial setting, and by the same argument as the base case of our induction, $RSet(u)$ will contain the set $vertex(leafexits(T^s_{win}))$.
2. u is in Z. In this case, again, T^s_{win} will always be a trivial 1 done tree. The set $RSet(u)$ will always be $\{\{\}\}$, and so $RSet(u)$ will always be equal to $vertex(leafexits(T^s_{win}))$.
3. u is a non-exit 0-vertex of component A_i. s must have one child s' in the strategy tree T^s_{win}. If $s \to s'$ is a move of T_A within A_i (i.e., $vertex(s)$ is not a call), then by the inductive claim $RSet(vertex(s'))$ will eventually contain a subset of $vertex(leafexits(T^{s'}_{win}))$. Thus, by rule ($a$) of the algorithm, $RSet(u)$ will "one update later" also contain a set $Y \subseteq vertex(leafexits(T^{s'}_{win}))$, but since $leafexits(T^s_{win}) = leafexits(T^{s'}_{win})$, we have what we want. If, on the other hand, the move from $s \to s'$ was a call meaning $e = vertex(s')$ is an entry of A_j, while $u = (b,e)$ is a call, then by induction $RSet(e)$ will eventually contain a $Y \subseteq vertex(leafexits(T^{s'}_{win}))$. Consider this $Y = \{ex_1,\ldots,ex_c\} \subseteq Ex_j$,

[1] There could be multiple occurrences of the same state s of T_A in T_{win}, but we assume that s is being somehow identified uniquely. We could do this by providing, together with s, the specific "path name" to the node s in T_{win}.

and consider $leafexits(T^{s'}_{win}) = \{s_1, \ldots, s_c\}$ in the tree T_{win}. Each s_i has exactly one child s'_i such that $vertex(s'_i) = (b, ex_i)$. Since $T^{s'_i}_{win}$ is a proper subtree of T^s_{win}, by induction each $RSet((b, ex_i))$ will eventually contain a set $Y_i \subseteq vertex(leafexits(T^{s'_i}_{win}))$. Now, using the fact that eventually, $RSet(e)$ will contain Y and that each $RSet((b, ex_i))$ will contain Y_i, we can use rule (c) to obtain that "one update later" $RSet((b, e))$ will contain a subset of $\cup_i vertex(leafexits(T^{s'_i}_{win}))$, which itself is a subset of $vertex(leafexits(T^s_{win}))$.

4. u is a non-exit 1-vertex of component A_i. In this case, without loss of generality, we can assume the only possibility is that s must have children $s'_1, \ldots, s'_d$ in the strategy tree T_{win}, and all $s \to s'_i$ moves are moves of T_A within A_i (i.e., not a call,, because call moves have only 1 successor in T_A, and hence calls can be viewed as 0-vertices). Then by the inductive claim $RSet(vertex(s'_i))$ will eventually contain a subset Y_i of $vertex(leafexits(T^{s'_i}_{win}))$. Thus, by rule (b) of the algorithm, $RSet(u)$ will "one update later" also contain a set $Y \subseteq \cup_i vertex(leafexits(T^{s'_i}_{win}))$, but since $leafexits(T^s_{win}) = \cup_i leafexits(T^{s'}_{win})$, we are done.

□

Let $maxEx = \max_i |Ex_i|$. For an upper bound on the running time of the algorithm, observe that each $RSet(u)$ gets updated at most 2^{maxEx} times. Each rule application can be done in time at most $2^{O(m \cdot maxEx)}$, where m is the maximum number of "neighbouring" vertices v', for any vertex v, such that $RSet(v)$ depends directly on $RSet(v')$ in some rule (the time taken for rule updates can be heavily optimized with good use of data structures common in program analysis). m is clearly upper bounded by the number of vertices, but is typically much smaller. There are $|A|$ vertices, so the worst case running time will be $|A| \cdot 2^{O(m \cdot maxEx)}$. However, observe that this worst-case analysis can be very pessimistic, as the retained minimal sets may converge to a fixpoint well before $RSet(v)$'s ever grow large. That is the principle advantage, and hope, offered by our equational approach. Of course, it will require much experimentation to determine under what circumstances this advantage materializes.

4 Algorithm for the Büchi Case

In the algorithm for Büchi conditions, the set-of-sets $BSet(u)$ that we associate with each vertex will be more elaborate than just subsets of the exits. Recall $maxEx$ is the maximum number of exits of any component in A. Let $Calls$ be the set of calls, i.e., call vertices (b, e), in the RGG. Let $GoodVertices = F \cup Calls$, be the union of the set of accept vertices (nodes) and call vertices in A. Let $maxMeasure = (|GoodVertices| \cdot 4^{maxEx}) + 1$.

Lets briefly sketch the intuition for the algorithm and proof. For each vertex u of A_i, $BSet(u)$ will contain sets of the form: $\{(\perp, m), (ex_1, tval_1), \ldots, (ex_k, tval_k)\}$, where $m \leq maxMeasure$, where each $ex_j \in Ex_i$, and where each $tval_j \in \{true, false\}$. The set can be interpreted to mean that player 0 can play starting at $\langle u \rangle$ in such a way that, no matter what player 1 does, we either will visit an accept node m times during our play, or else we will reach a state $\langle ex_j \rangle$, and we will do so having visited an accept node along the way if $tval_j = true$. The rules will be used to update these sets in a consistent way. What we will show is that $\{(\bot, maxMeasure)\}$ enters $BSet(u)$ iff there is a finite strategy tree rooted at $\langle u \rangle$ such that on every path in the strategy tree we necessarily repeat a vertex, having visited an accept state in between visits, with the stack getting no smaller in between visits, and such that we are in the same "context of obligations" (to be made precise) in both visits. This allows us to repeatedly apply the same substrategies in order to achieve an infinite winning strategy tree.

Let $P_i = \{(ex, tval) \mid ex \in Ex_i \;\&\; tval \in \{true, false\}\}$. Let $J = \{(\bot, m) \mid 1 \leq m \leq maxMeasure\}$. We say a set $X \in 2^{P_i \cup J}$ is *well-formed* if both the following conditions hold: (1) For all $ex \in Ex_i$, $(ex, true) \notin X$ or $(ex, false) \notin X$ (or both). (2) If $(\bot, m) \in X$ then for all $m' \neq m$, $(\bot, m') \notin X$. Let τ_i be the set of of all well-formed sets in $2^{P_i \cup J}$. We will refer to $X \in \tau_i$ as a set of type τ_i, and similarly we will refer to sets-of-sets $\mathcal{S} \subseteq \tau_i$ as having type τ_i. For $S, S' \in \tau_i$, we write $S' \preceq S$ iff $(a), (b), (c)$ hold: (a) if for $ex \in Ex_i$, $((ex, false) \notin S$ and $(ex, true) \notin S)$, then $((ex, false) \notin S'$ and $(ex, true) \notin S')$. (b) if $(ex, true) \in S$, then $(ex, false) \notin S'$. (c) if $(\bot, j) \in S$ then $(\bot, j') \in S'$ such that $j' \geq j$.
For a set $\mathcal{S} \subseteq \tau_i$ let $Min(\mathcal{S})$ denote the set of those sets $X \in \mathcal{S}$ that are minimal with respect to $\preceq$ in $\mathcal{S}$. Given a set X of type τ_i, let $Increment(X)$ be the smallest set such that: if $(ex, tval)$ is in X, then $(ex, true)$ is in $Increment(X)$, and if $(\bot, j)$ is in X, then $(\bot, min(j+1, maxMeasure))$ is in $Increment(X)$. Extending the notation, for $\mathcal{S} \subseteq \tau_i$, let $Increment(\mathcal{S}) = \{Increment(X) \mid X \in \mathcal{S}\}$.

For sets $X_1, \ldots, X_k$, each of the same type τ, we define a "boxy union" $X' = \bigsqcup_{i \in \{1,\ldots,k\}} X_i$ to be a subset of $X = \bigcup_{i \in \{1,\ldots,k\}} X_i$ as follows: if element $(ex, true)$ and $(ex, false)$ are both in X, then only $(ex, false)$ is in X'. Moreover, there is a $(\bot, j')$ in X' if j' is the minimum value j for which $(\bot, j)$ is in X. Intuitively, "boxy union" reflects choices optimal for the adversary (player 1).

We will associate to each $u \in Q_i$ a set $BSet(u) \subseteq \tau_i$, such that $BSet(u)$ will eventually contain $\{(\bot, maxMeasure)\}$ if and only if player 0 has a winning strategy in the Büchi game. Let F be the set of accepting nodes.
Initialization: For each $z \in F$, initially $\{(\bot, 1)\} \in BSet(z)$. Moreover, for each $x \in Ex_i$, initially $BSet(x)$ contains $\{(x, tval)\}$, where $tval = true$ if $x \in F$, and otherwise $tval = false$. For all other vertices v, we initialize $BSet(v) := \emptyset$.
Rule application: we make sure the following relationships hold:

1. For every 0-vertex v, with the exception of exits or calls, or nodes in F:
$$BSet(v) := Min(\bigcup_{(v,w) \in \delta} BSet(w))$$
2. For every 0-vertex v that isn't a exit or call, but is in F,
$$BSet(v) := Min(Increment(\bigcup_{(v,w) \in \delta} BSet(w)) \cup \{\{(\bot, 1)\}\})$$

3. For every 1-vertex v with successors $\{w_1, ..., w_k\}$ (v not an exit or call), such that v is not in F:

$$BSet(v) := Min(\{\bigsqcup_{i \in \{1,...,k\}} X_i \mid \forall i : X_i \in BSet(w_i)\})$$

4. For every 1-vertex v with successors $\{w_1, ..., w_k\}$ (v not an exit or call), such that v is in F:

$$BSet(v) := Min(Increment(\{\bigsqcup_{i \in \{1,...,k\}} X_i \mid \forall i : X_i \in BSet(w_i)\}) \cup \{\{(\bot, 1)\}\})$$

5. For a call (b, e_i) in component A_j, where e_i is an entry point of A_i, and where A_i has exits $\{x_1, ..., x_k\}$,
$BSet((b, e_i)) :=$

$$Min(\{((\bigsqcup_{x \in X} Incr_{X,x}(X_{b,x})) \sqcup D_X) | X \in BSet(e_i) \ \& \ X_{b,x} \in BSet((b, x))\})$$

where $D_X = \{(\bot, j) \mid (\bot, j) \in X\}$, and where $Incr_{X,x}(X')$ is equal to X' if there is an element $(x, false)$ in X, and otherwise is $Increment(X')$.

We call a rule application that changes the value of some $BSet(v)$, an *update.*

Theorem 2. *The algorithm halts after at most* $|A|^2 \cdot 2^{O(maxEx)}$ *updates to each* BSet(v)*. When it halt,* $\{(\bot, maxMeasure)\} \in$ BSet(u) *if and only if player* 0 *has a winning strategy in the Büchi game* B_A *starting at* $\langle u \rangle$*.*

For the proof please see the appendix. For the worst-case time complexity: each update can be carried out in time at most $|A|^{O(1)} \cdot 2^{O(m \cdot maxEx)}$, where m is the maximum number of "neighbouring" vertices of any vertex. So the total running time is $|A|^{O(1)} \cdot 2^{O(m \cdot maxEx)}$. The algorithm as described will always require at least $maxMeasure$ updates to reach $\{(\bot, maxMeasure)\}$ in some $BSet(v)$. The algorithm can be reformulated to avoid this. We omit such a reformulation.

5 Conclusions

We have provided alternative algorithms, using second-order data flow equations, for determining whether a player has a winning strategy on recursive game graphs, a model that is expressively equivalent to pushdown games. Our algorithms generalize the approach of using Datalog rules for analysis of recursive state machines from [AEY01], as in [ATM03b], and they can also be viewed as a "automaton-free" version of the algorithms given by [Cac02a] for pushdown games. Several extensions of Cachat's work have appeared in more recent literature. [CDT02] extends the algorithms to check properties such as "stack boundedness" of infinite plays (a notion which was studied for runs of recursive state machines in [AEY01]). Also, [Ser03,Cac02b] extends the work to games with parity conditions. It may be possible to carry out these extensions in our equational framework, but we have not done so here.

Acknowledgements. Thanks to R. Alur, P. Madhusudan, and M. Yannakakis for extensive discussions and helpful comments.

References

[AEY01] R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *Proc. of CAV'01*, pp. 304–313, 2001.

[App98] A. Appel. *Modern compiler implementation*. Cambridge U. Press, 1998.

[ATM03a] R. Alur, S. La Torre, and P. Madhusudan. Modular strategies for infinite games on recursive graphs. In *CAV'03*, *LNCS* vol. 2725, pages 67–79, 2003.

[ATM03b] R. Alur, S. La Torre, and P. Madhusudan. Modular strategies for recursive game graphs. In *TACAS'03*, *LNCS* vol. 2619 , pp. 363–378, 2003.

[BEM97] A. Boujjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: App's to model checking. In *CONCUR'97*, pages 135–150, 1997.

[BGR01] M. Benedikt, P. Godefroid, and T. Reps. Model checking of unrestricted hierarchical state machines. In *ICALP'01*, *LNCS* 2076, pp. 652–666, 2001.

[BR00] T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN'2000*, volume 1885 of *LNCS*, pages 113–130, 2000.

[BS92] O. Burkart and B. Steffen. Model checking of context-free processes. In *Proc. of CONCUR*, volume 836 of *LNCS*, pages 123–137, 1992.

[Cac02a] T. Cachat. Symbolic strategy synthesis for games on pushdown graphs. In *Proc. of ICALP*, volume 2380 of *LNCS*, 2002.

[Cac02b] T. Cachat. Uniform solution of parity games on prefix recognizable graphs. In *Infinity 2002, 4th. Int. Workshop*, 2002.

[Cau90] D. Caucal. On the regular structure of prefix rewriting. In *5th Coll. on Trees in Algebra and Programming*, *LNCS* vol. 431, pp. 87–102, 1990.

[CDT02] T. Cachat, J. Duparc, and W. Thomas. Solving pushdown games with a $sigma_3$ winning condition. In *CSL'02*, pp. 322–336, 2002.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *12th Computer-Aided Verification*, volume 1855 of *LNCS*, pages 232–247. Springer, 2000.

[FWW97] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. In *Infinity'97 Workshop*, volume 9 of *Electronic Notes in Theoretical Computer Science*, 1997.

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer, 1999.

[Rep98] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, 1998.

[Ser03] O. Serre. Note on winning strategies on pushdown games with omega-regular winning conditions. *Information Processing Letters*, 85(6):285–291, 2003.

[Tho95] W. Thomas. On the synthesis of strategies in infinite games. In *STACS*, volume 900 of *LNCS*, pages 1–13, 1995.

[Wal96] I. Walukiewicz. Pushdown processes: games and model checking. In *Computer-Aided Verification*, pages 62–75, 1996.

A Proof of Correctness for Büchi Case

Proof. (of Theorem 2) Due to space we state several lemmas without proof. First, ($\Rightarrow$). We show that if $\{(\perp, maxMeasure)\}$ ends up in $BSet(u)$ then Player 0 has a winning strategy in the game (this is the harder direction). Suppose $S = \{(\perp, maxMeasure)\}$ does end up in $BSet(u)$. Our first step is to construct a "witness" to this in the form of a straight-line program, W, using the sets $BSet(v)$. A witness W for u has the form: $(1, C_1, Pre_1)(2, C_2, Pre_2)\ldots(l, C_l, Pre_1)$. It consists of l *lines* of the form (d, C_d, Pre_d), with d the *line number*, and:

- C_d is the *content*, and has the form (v, S) where $v \in Q_i$ is a vertex of A_i, $S \in \tau_i$, and there exists $S' \in BSet(v)$ such that $S' \preceq S$.
  If $(\perp, m) \in S$, we say line d "contains a measure", and its measure is m.
- Pre_d is a (possibly empty) list $[d_1, \ldots, d_k]$ of predecessor line numbers, with $d_i < d$ for each i.

Morever:

- No two lines have the same content, thus content determines line number.
- A line with no predecessors is called an *initial* line. If the d'th line is initial, then it must be either of the form: $(d, (v, \{(\perp, 1)\}), [])$, where $v \in F$, or of the form $(d, (ex, \{(ex, tval)\}), [])$ where ex is an exit, and $tval$ is true if and only if $ex \in F$.
- If the d'th line has predecessors, it has the form $(d, (v, S), [d_1, \ldots, d_i])$, where:
  - The content (v, S) is *implied in one step* by the content of its predecessors. By this we mean that if the contents of the d_j are, respectively, (v_j, S_j), then there is a rule $\mathcal{R}$ such that if $S_j \in BSet(v_j)$ for each j, then one application of $\mathcal{R}$ would put $S' \in BSet(v)$, where $S' \preceq S$.
    E.g., if $v = (b, e)$ is a call, and $S = \{(\perp, m), (ex_1, tval_1), \ldots, (ex_k, tval_k)\}$, then d_1 will be a line with content (e, S_1) and d_j, for $j > 1$, will have content $((b, ex'_j), S_j)$, such that $(ex'_j, tval)$ is in S_1, moreover so that $S = \bigsqcup_{j\in\{2,\ldots,i\}} S_j \sqcup \{(\perp, m') \mid (\perp, m') \in S_1\}$.
  - Moreover, the measures $(\perp, m')$ in any of the content sets $S_1, \ldots, S_i$, of lines $d_1, \ldots, d_i$, should be as weak as possible in order to imply S, meaning that it should not be possible to decrease the measure in any single S_j and still imply S in one step.
- The last line, line l, is of the form $(l, (u, \{(\perp, maxMeasure)\}), Pre_l)$.

Lemma 1. *If* $\{(\perp, maxMeasure)\} \in \mathrm{BSet}(u)$ *then a witness* W *for* u *exists.*

The lemma can be proved by induction on the number of rule applications it took for $\{(\perp, maxMeasure)\}$ to enter $BSet(u)$. We will use W to define a winning strategy tree for player 0. To do this, we will first need some facts about W. By a *path* from a line d to a line d', $d > d'$, in W, we mean a sequence $\gamma = d_1, \ldots, d_n$, such that $d = d_1$, $d' = d_n$, and $d_{i+1} \in Pre_{d_i}$ for all $i \in \{1, \ldots, n-1\}$. (Note: the path goes from higher to lower line numbers.)

Lemma 2. *On any path $\gamma = d_1, \ldots, d_n$ in W, the measure is non-increasing, meaning, for $i < j$, if line d_i contains a measure m then line d_j either contains a measure $m' \leq m$, or does not contain a measure, and if line d_i does not contain a measure then neither does line d_j.*

Proof. A line has a measure iff any of its predecessors has a measure. Moreover, because of the minimality constraint on predecessor measures in W, the measure of a line $(d, (v, S), Pre_d)$ is the same as that of its predecessors that contain a measure unless v is either a call or an accept node (i.e., a good vertex), in which case the measure can be 1 greater. The measure is therefore non-increasing. □

For a path γ in W, let $goodLines(\gamma)$ be the number of lines of γ of the form $(d, (v, S), Pre_d)$ where $v \in GoodVertices$, i.e., v is either a call vertex or an accept node. Let d and d' be two lines such that there is a path from d to d' in W, and let $goodDist_W(d, d') = \min\{goodLines(\gamma) \mid \gamma \text{ is a path from } d \text{ to } d' \text{ in } W\}$. For two lines $(d, (v, S), Pre_d)$ and $(d', (v', S'), Pre_{d'})$ that both contain a measure, we say the contents C_d and C'_d are *the same except for the measure* if $v = v'$, and S and S' differ only by the fact that $(\perp, m) \in S$, and $(\perp, m') \in S'$, with $m \neq m'$. In such a case, we write $C_d \ll C'_d$, if $m < m'$.

Lemma 3. *Let d be any initial line of W of the form $(d, (v, \{(\perp, 1)\}), [])$ and let l be the last line number, then*

1. $\text{goodDist}_W(l, d) \geq maxMeasure$.
2. *On any path $\gamma = l = d_r d_{r-1} \ldots d_1 = d$ from l to d, there exist two distinct lines d_i and d_j, $i > j$, such that $C_{d_j} \ll C_{d_i}$.*

Proof. (1) The measure at l is $maxMeasure$, while the measure at d is 1. Thus, in a path γ from line l to d, since we must have $maxMeasure - 1$ opportunities to decrement the measure, and can only do so on good lines, and since line d is also a good line, we must encounter at least $maxMeasure$ good lines.
(2) There are at most $(|GoodVertices| \cdot 4^{|maxEnt|})$ different contents (v, S) where v is a good-vertex (counting only once contents that the same except for the measure). Thus, since $maxMeasure = (|GoodVertices| \cdot 4^{|maxEnt|}) + 1$, by the pigeon-hole principle and by (1), there must be two such lines d_i and d_j in any γ. Since the measure is non-increasing, we must have $C_{d_j} \ll C_{d_i}$. □

We now use W to construct a strategy tree for player 0. Each node of the strategy tree will be labelled by a triple $(s, d, Stack)$, where:

- s is a global state $\langle \bar{b}, v \rangle$ of B_A.
- d is a line number in the witness straight-line program W.
- $Stack$ is a stack $[\beta_j, \beta_{j-1}, \ldots, \beta_1]$, where each β_r defines a mapping from a subset of the exits of a component to line numbers in W (in a consistent way to be defined).

The root of the strategy tree will be labelled by $(\langle u \rangle, l, [])$, (where l is the last line of W). Thereafter, if a node is labelled by $(\langle \bar{b}, v \rangle, d, Stack)$, we use W to construct its children. Suppose, for example, line d of W is $(d, (v, S), [d_1, d_2, \ldots, d_i])$, and suppose $v = (b, e)$ is a call. Let line d_1 have content (e, S_1). Then we create one child for the node and label it by the tuple $(\langle \overline{b}, b, e\rangle, d_1, push(\beta, Stack))$, where β is a mapping from each ex_j, such that $(ex_j, tval_j)$ is in S_1, to a line d_{i_j} whose content is of the form $((b, ex_j), S_j)$. In other words, the element pushed on the Stack tells us where in the witness program to return to when returning from b on the call stack, as dictated by the predecessors $d_2, \ldots, d_i$ in line d. For other kinds of vertices v, we can construct corresponding children of nodes in the strategy tree. If in the strategy tree so constructed we reach a state $\langle \overline{b}, b, ex\rangle$, whose vertex ex is an exit, then we pop *Stack* and use its contents to dictate what the children of that node should be labelled, using the state content to assign the line number of W to the triple associated with the children.

Lemma 4. *The construction outlined above yields a finite tree T_W, all of whose leaves are labelled by triples of form $(\langle \overline{b}, v\rangle, d, Stack)$, where line d is an initial line of W with content $(v, \{(\bot, 1)\})$.*

The lemma can be established using the structure of W. The key point is that if we ever reach an exit following the program W, we know that our Stack contains a return address where we may continue to build T_W. We want to build from T_W an infinite strategy tree.

Lemma 5. *For any leaf z in T_W labelled by $(\langle \overline{b}, v\rangle, d, Stack)$, the root-to-leaf path in T_W must include a subsequence $H_{maxMeasure}, \ldots, H_1$, where $H_i = (\langle \overline{b_i}, v_i\rangle, d_i, Stack_i)$, and where*

- *v_i is a good vertex of A, for each i,*
- *each $\overline{b_i}$ is a prefix of $\overline{b}$, and every node on the path from H_i to z contains $\overline{b_i}$ as a prefix of its call stack (and hence the Stack at every such node also contains as a prefix the stack $Stack_i$ at H_i).*
- *H_i has (v_i, S_i) as the content of its line, such that $(\bot, i) \in S_i$.*

In other words, the subsequence $H_{maxMeasure}, \ldots, H_1$ of nodes along the root to leaf path in T_W will witness the decrementation of the counter from $maxMeasure$ down to 1. The counter can only be decremented by 1 at a time, and so all such distinct witnesses must exist. Now, because of the size of $maxMeasure$, there must be two distinct H_r and $H_{r'}$, $r' < r$, with labels $(\langle \overline{b_r}, v\rangle, d_r, Stack_r)$ and $(\langle \overline{b_{r'}}, v\rangle, d_{r'}, Stack_{r'})$, such that they both have the same vertex v (which must be a good vertex, either an accept node or a call), and such that the lines d_r and $d_{r'}$ have content (v, S) and (v, S'), where S and S' are exactly the same except that $(\bot, r)$ is in S while $(\bot, r')$ is in S'. We mark any such node $H_{r'}$. We then eliminate the subtrees rooted at all marked nodes in T_W. This yields another finite tree T'. We will use T' to construct an infinite strategy tree $T^\star$ that is accepting for player 0. Let M be the set of (good) vertices v such that a state $\langle \overline{b}, v\rangle$ labels some leaf of T'.

Lemma 6. *For any leaf of T' labelled $L = (\langle \overline{b}, v\rangle, d, Stack)$, there is a finite strategy tree T_L for player 0, with root labelled L, such that the state labels on every root-to-leaf path contain $\overline{b}$ as a stack prefix, and such that all leaf labels $(\langle \overline{b'}, v'\rangle, d', Stack')$, have $v' \in M$. Moreover, every root-to-leaf path in T_L contains an accept state.*

Using the lemma we can incrementally construct $T^\star$ from T'. We repeatedly attaching a copy of T_L to every leaf labelled L in the tree T'. Since the process always produces leaves labelled s whose node v is in M, and since the stacks can only grow, we can extend the tree indefinitely. Every path will contain infinitely many accept states, because each finite subtree that we attach contains an accept state on every root to leaf path.

($\Leftarrow$): If Player 0 has a winning strategy in the Büchi game starting at $\langle u \rangle$, then $\{(\perp, maxMeasure)\}$ will eventually enter $BSet(u)$. We omit the proof, which is similar to the proof that if player 0 has a winning strategy in the reachability game from $\langle u \rangle$, then $\{\}$ will eventually enter $RSet(u)$. $\square$