Interface Compatibility Checking for Software Modules*

Arindam Chakrabarti*, Luca de Alfaro**, Thomas A. Henzinger*,
Marcin Jurdziński*, and Freddy Y.C. Mang***

*EECS, University of California, Berkeley.
**CE, University of California, Santa Cruz.
***Advanced Technology Group, Synopsys Inc.

Abstract. We present a formal methodology and tool for uncovering errors in the interaction of software modules. Our methodology consists of a suite of languages for defining software interfaces, and algorithms for checking interface compatibility. We focus on interfaces that explain the method-call dependencies between software modules. Such an interface makes assumptions about the environment in the form of call and availability constraints. A call constraint restricts the accessibility of local methods to certain external methods. An availability constraint restricts the accessibility of local methods to certain states of the module. For example, the interface for a file server with local methods `open` and `read` may assert that a file cannot be read without having been opened. Checking interface compatibility requires the solution of games, and in the presence of availability constraints, of pushdown games. Based on this methodology, we have implemented a tool that has uncovered incompatibilities in TinyOS, a small operating system for sensor nodes in adhoc networks.

1 Introduction

In structured software design, functionality and data is arranged in software modules. Each module has a set of procedures, or *methods*, for accessing the encapsulated data. Modules can be treated as components, for example, taken from libraries, or implemented by different vendors. This raises the question of when two modules are *compatible*. A limited answer to this question is given by traditional type systems. For example, if method `a` of module A calls method `b` of module B, then the number and types of the actual parameters of the call of `b` in A must match the number and types of the formal parameters of the implementation of `b` in B; otherwise, the modules A and B are incompatible. This weak form of compatibility is resolved by type checking. Note that the type of an external method call in module A, say `b(n:int)`, is an *assumption* about the environment of A, namely, that it provides an implementation of `b` with a single formal parameter, which is an integer. The assumption is checked when the environment is provided.

* This research was supported in part by the AFOSR grant F49620-00-1-0327, the DARPA grant F33615-00-C-1693, the MARCO grant 98-DT-660, the NSF grants CCR-9988172, CCR-0085949, CCR-0132780, the SRC grant 99-TJ-683, and the Polish KBN grant 7-T11C-027-20.

We define and check two stronger forms of software module (SM) compatibility. The first is called *stateless SM interface compatibility*. Stateless SM interfaces can express assumptions about the call graph of the environment. For example, a common design requirement is that an initialization method must not call itself recursively. Consider again module A, this time with the two local methods `a-init` and `a-update`. Suppose that the implementation of `a-init` calls the external method `b`, and the designer of A wants to make sure that whatever `b` does, it causes no recursive call-back of `a-init`. This constraint about the environment can be written as a *call assumption* for module A, namely, `b:not{a-init}`. In general, the call assumption `b:not{a_1,...,a_k}` for a module A with external method `b` and local methods `a_1`, ..., `a_k` has the following interpretation: every chain of method calls that can be caused by an invocation of `b` must *not* contain any method in `{a_1,...,a_k}`. In particular, the default call assumption for an external method `b` is `b:not{}`; this does not constrain the implementation of `b`.

A second, even stronger form of software module compatibility is called *stateful SM interface compatibility*. Stateful SM interfaces can express assumptions about the state of the module when local methods are called by the environment. For example, a common design requirement is that an update method must not be called before the corresponding initialization method is called. Consider again module A with the local methods `a-init` and `a-update`. Suppose that the state variable x of A records whether or not the method `a-init` has been called: initially $x = 0$, and x changes to 1 with the first call of `a-init`, where it remains. The constraint that the environment calls `a-update` only when $x = 1$ can be written as a pair of *availability assumptions* for module A, namely, `a-init:true` and `a-update:x=1`, with the following interpretation: when $x = 0$, then only `a-init` may be called by the environment; when $x = 1$, then both `a-init` and `a-update` may be called. In general, the availability assumption `a:p` for a module A asserts that the local method `a` of A can be called only when the predicate `p` is true in the current state of A. The default availability assumption for a local method `a`, which does not constrain the environment, is `a:true`.

It is obviously undecidable to check if call and availability assumptions for module A are satisfied by the *code* for environment module B.[1] We therefore require the designer of a module to explicitly provide an interface. An SM interface makes *guarantees* about how the local methods interact with the environment, in addition to the assumptions about how the environment is expected to interact with the local methods. Then, two modules A and B are compatible if the guarantees of A meet the assumptions of B, and vice versa. Consider again the stateless case, where the interface of module A makes the call assumption `b:not{a-init}`. If the interface of module B, which owns the method `b`, provides the call guarantee `b:{}`, meaning that the implementation of `b` calls no other methods, then the modules A and B are compatible. On the other hand, if the interface of B has the call guarantee `b:{a-init}`, meaning that the implementation of `b` calls `a-init`, then the modules A and B are incompatible.

[1] Static analysis may be used for conservative estimations.

The interesting case is the third possibility, that the implementation of `b` calls some methods other than `a-init`. Suppose that the interface of B provides the call guarantee `b:{c}`, meaning that the implementation of `b` calls the new method `c`, which is external to both modules A and B. In this case, the call assumption `b:not{a-init}` of A may or may not be satisfied, depending on whether or not the implementation of `c` calls (directly or indirectly) `a-init`. While a pessimistic approach would reject the composition of A and B, because compatibility is not ensured for *all* environments —i.e., implementations of `c`— we instead take the optimistic approach to compatibility [6] and compute the derived call assumption `c:not{a-init}` for the combined interface of $A||B$. In other words, A and B are considered compatible, because there is *some* environment that makes A and B work together properly, namely, the environment that implements `c` without calling `a-init`. Note that only the chosen, optimistic approach to compatibility is associative. Suppose that a third module C provides an implementation of method `c` with call guarantee `c:{}`; that is, `c` calls no further methods. Then the composition $A||B||C$ is well-formed. While the optimistic approach permits all ways of assembling this system, namely, $(A||B)||C$ and $A||(B||C)$ and $(A||C)||B$, the pessimistic approach rejects the first one.

The optimistic approach to compatibility is made possible by the ability of interfaces to express environment assumptions, which can then be propagated when composing interfaces. In the stateless case, SM interface compatibility checking, as well as the derivation of call assumptions for the composite interface, are graph problems that can be solved in quadratic time. If state is involved, checking optimistic compatibility between two interfaces A and B requires the solution of a two-player *game* [9, 6, 7]. Player 1 represents both A and B, and player 2 represents the environment. If player 2 has a strategy of satisfying the call and availability assumptions of both A and B, then the two interfaces are compatible (because there is a "helpful" environment); otherwise they are incompatible. Note that if the composition of A and B is *closed*, i.e., calls no external methods, then the game disappears, and compatibility checking simply resolves the assumptions of A against the guarantees of B, and vice versa. As call chains may be recursive, we need to solve *pushdown* games, rendering compatibility checking and composition for stateful SM interfaces exponential. However, SM interfaces tend to be much smaller than the underlying module implementations: for instance, in the stateful example from above, the interface has only two states ($x = 0$ and $x = 1$), whereas the state of module A itself may be arbitrarily complex (depending on which data structures A contains).

We have modified the JBuilder programming environment to permit the annotation of Java objects with interfaces. When defining a Java object, the programmer may specify its interface as commentary, and our tool automatically checks its compatibility with the interfaces of other objects that have already been defined. The tool implements the compatibility check for the interface automata of [6], as well as the stateless and stateful SM interfaces presented here. Interface automata are based on finite-state games, and do not support recursive call-backs between modules. This is insufficient for many software applications,

including our software driver TinyOS [13], a small operating system for sensor nodes in adhoc networks. TinyOS is structured into six modules, which represent different service layers. We have defined stateful SM interfaces for two of the layers, and discovered two incompatilities in their interaction. The notion of interface state that needs to be considered in this example is considerably less complex than the full state of the implementation, bearing out the promise of our methodology. In other words, by aiming at certain limited but common classes of module interaction errors, rather than intra-module errors, we are able to avoid many of the obstacles to fully automatic and complete software verification.

We are not the first to propose a formalization of software module interfaces. Many researchers have addressed this issue by developing languages for writing software specifications and contracts, e.g., [16, 15, 17, 5, 11]. These languages are typically based on pre- and postconditions, and therefore related in expressiveness to stateful SM interfaces (an availability assumption is a restricted kind of precondition, which cannot refer to the parameters of method calls). The key difference between our interfaces and software contracts is that contract violations are detected at run-time, while interface incompatibilities are uncovered at compile-time. In this respect, interfaces are *types*, and the stateful SM interfaces are related to recent trends in type systems to capture behavioral aspects of software [12, 8, 14], and type systems for module interaction [1]. Indeed, the latter also advocates a game-theoretic view. Architecture description languages (e.g., [3]) and software modeling languages (e.g., UML) also support various degrees of formality in specifying module interactions, but to our knowledge, none of these languages capture the optimistic, game-based approach to compatibility.

2 Stateless Software Module Interfaces

A *stateless SM interface* $I = (M^L, M^E, \mathcal{C}, \mathcal{B})$ consists of the following:

- A set M^L of *local methods*. These are the methods that are defined within the module.[2]
- A set M^E of *external* (or *imported*) *methods*. These are the methods that are not defined within the module, but called from method definitions within the module. We write M for the set of all methods known by the interface, i.e., we define $M = M^L \cup M^E$.
- A set $\mathcal{C} = \{C(l) \mid l \in M^L\}$ of *call guarantees*. For each local method $l \in M^L$, the call guarantee $C(l) \subseteq M$ specifies the (local and external) method calls that occur in the definition of l.
- A set $\mathcal{B} = \{B(m) \mid m \in M\}$ of *call assumptions*. For each method $m \in M$, the call assumption $B(m) \subseteq M^L$ specifies the local methods whose execution is forbidden as a result of an invocation of m.

Note that call guarantees refer only to direct calls, that is, $m \in C(l)$ asserts that m is called directly by the definition of l. Call assumptions, on the other

[2] We assume that all local methods can be called by other modules. It is straightforward to further differentiate between *hidden* and *exported* local methods.

hand, refer to direct as well as indirect calls, that is, $l \in B(m)$ asserts that there must not be a sequence $m_0, m_1, \ldots, m_k$ of (local, known external, or unknown external) methods such that $m_0 = m$ and $m_k = l$ and for all $0 \leq i < k$, the definition of m_i calls m_{i+1}. Call guarantees and call assumptions may be inconsistent. For example, for local l and m, it must not happen that both $m \in C(l)$ and $m \in B(l)$. Consistency is defined formally below.

Call guarantees must conform with the module implementation[3]; call assumptions are constraints that the module designer puts on the implementation of external methods. For example, for local l and external e, if $l \in B(e)$, then e is expected to be implemented in a way that does not cause a direct or indirect call of l. Similarly, for local l and m and external e such that $e \in C(m)$, that is, e is called by m, if $l \in B(m)$, then e is again expected not to cause a direct or indirect call of l. In the latter case, the call assumption $l \in B(e)$ can be derived. If an interface contains all derived call assumptions, then it is called complete.

Consistent and complete interfaces. For a local method $l \in M^L$ and a method $m \in M$, we say that l *calls* m, and write $m \in C^*(l)$ if there is a sequence $m_0, m_1, \ldots, m_k$ of methods $m_i \in M$ such that $m_0 = l$ and $m_k = m$, and $m_{i+1} \in C(m_i)$ for all $0 \leq i < k$. Note that m_i must be local for all $i < k$, and m_k may be either local or external. We say that l *properly calls* m, and write $m \in C^+(l)$, if $k > 0$. For a method $m \in M$ and local method $l \in M^L$, we say that m *must not call* l if $l \in B(m)$. The call assumption $B(m)$ of a method $m \in M$ is *complete* in the interface I if for all methods $m' \in M$, and all local methods $l, l' \in M^L$, if l must not call m', and l calls m, and l' calls m', then m must not call l'. The interface I is *complete* if the call assumptions of all methods in M are complete in I. The *completion* $I^c = (M^L, M^E, \mathcal{C}, \mathcal{B}^c)$ of I is the (unique) stateless SM interface whose set $\mathcal{B}^c$ of call assumptions is the component-wise smallest family of sets $B^c(m)$ such that for all methods $m \in M$, both $B(m) \subseteq B^c(m) \subseteq M^L$ and $B^c(m)$ is complete in I^c.

The call assumption $B(l)$ of a local method $l \in M^L$ is *satisfied* in the interface I if l does not properly call any method that it must not call. The interface I is *consistent* if the call assumptions of all local methods in M^L are satisfied in I. Note that consistency can be checked in time required to compute the transitive closure C^+ of C, which is quadratic in the size of the interface (i.e., in the number of edges of a graph). Also note that for consistency, it does not matter whether I is complete, because if some call assumption $B^c(l)$ is not satisfied in the completion I^c, then there is a local method $l' \in M^L$ such that the call assumption $B(l')$ is not satisfied in the original interface I. Hence we need not insist on completeness in stateless SM interfaces.

Interface compatibility and composition. Two stateless SM interfaces $I = (M_I^L, M_I^E, \mathcal{C}_I, \mathcal{B}_I)$ and $J = (M_J^L, M_J^E, \mathcal{C}_J, \mathcal{B}_J)$ are *proto-compatible* if their local methods are disjoint: $M_I^L \cap M_J^L = \emptyset$. If I and J are proto-compatible, then the *composition* of I and J is the stateless SM interface $I||J = (M^L, M^E, \mathcal{C}, \mathcal{B})$ with

- $M^L = M_I^L \cup M_J^L$ and $M^E = (M_I^E \cup M_J^E) \setminus M^L$;

[3] Indeed, it would not be difficult to derive call guarantees automatically from the module implementation by parsing method definitions.

– for all $l \in M^L$, we have $C(l) = C_I(l)$ if $l \in M_I^L$, and $C(l) = C_J(l)$ if $l \in M_J^L$;
– for all $m \in M$, we have $B(m) = B_I(m) \cup B_J(m)$.

Note that composition is associative. The stateless SM interfaces I and J are *compatible* if they are proto-compatible, and the composition $I||J$ is consistent.

Theorem 1. *The compatibility of two stateless SM interfaces can be checked in quadratic time.*

3 Background: Pushdown Games

A *labeled pushdown game* $\mathcal{G} = (Q, \Sigma, \sigma, \Gamma, c_0, \hookrightarrow)$ consists of the following:

– a finite set Q of (*control*) *states*, partitioned into the *existential* states $Q_\exists$ and the *universal* states $Q_\forall$;
– a finite *stack alphabet* Γ;
– an initial configuration $c_0 \in Q \times \Gamma^+$;
– a transition relation $\hookrightarrow \subseteq (Q \times \Gamma) \times (Q \times Cmd(\Gamma))$, where $Cmd(\Gamma) = \{skip, pop\} \cup \{push(\gamma) \mid \gamma \in \Gamma\}$.

The *game tree* $\mathcal{T}(\mathcal{G})$ is a the labeled tree of configurations of $\mathcal{G}$ with the root c_0 such that each vertex $(q, \gamma \cdot w) \in Q \times \Gamma^+$ has the following successors:

– $(q', \gamma \cdot w)$ if $(q, \gamma) \hookrightarrow (q', skip)$;
– (q, w) if $(q, \gamma) \hookrightarrow (q', pop)$;
– $(q', \gamma' \cdot \gamma \cdot w)$ if $(q, \gamma) \hookrightarrow (q', push(\gamma'))$.

Furthermore, the vertex (q, w) is existential if $q \in Q_\exists$, and universal otherwise.

A two-player game is played on a game tree as follows. The game starts at the root. At an existential vertex, the existential player chooses a successor, and at a universal vertex, the universal player chooses a successor. The *reachability problem* for pushdown games asks, given a labeled pushdown game $\mathcal{G}$ and a control state $f \in Q$, if the existential player has a strategy to direct the play on the game tree $\mathcal{T}(\mathcal{G})$ to a node (f, w), for some $w \in \Gamma^*$.

Theorem 2. [18] *The reachability problem for labeled pushdown games is complete for DEXPTIME.*

4 Stateful Software Module Interfaces

When equipping SM interfaces with state, we use interface programs to specify the state transitions. An interface program is an abstraction of the code implementing a method and must be provided by the software designer. We represent interface programs by *flow graphs*, whose nodes represent control locations and whose edges are labeled with instructions, including method calls [10]. Nondeterministic branching is allowed and can be the result of abstraction. More formally, for a set X of typed variables and a set M of methods, an *interface program* $P = (L, E, \vdash, \dashv, \mu)$ over X and M is a labeled graph, consisting of the following:

– A finite set L of *program locations*. Every program has an *initial location* $\vdash \in L$ and a terminal location $\dashv \in L$.

– A set of edges $E \subseteq V \times V$, and a labeling function μ: $E \to \mathit{Instr}(X, M)$, where $\mathit{Instr}(X, M)$ is the set containing assignment and conditional instructions over the variables in X, and calls to the methods in M.

We write $\mathit{Progs}(X, M)$ for the set of interface programs over X and M. A *stateful SM interface* $F = (X, M^L, M^E, \mathcal{P}, \mathcal{A}, \mathcal{B})$ consists of the following:

– A finite set $X = \{x_1 : D_1, \ldots, x_k : D_k\}$ of typed *interface variables*, where $D_1, \ldots, D_k$ are finite sets of values. We refer to the type-respecting valuations of the variables in X as *interface states*, and write $\mathit{States}(X)$ for the set of interface states.
– A finite set M of methods, partitioned into the set M^L of *local methods* and the set M^E of *external methods*.
– For every local method $l \in M^L$, an *interface program* $P(l) \in \mathit{Progs}(X, M)$ over the variables X and methods M. The interface program $P(l)$ abstracts the implementation of l by recording method calls, and the (nondeterministic) changes in the interface state between method calls.
– For every local method $m \in M^L$, an *availability assumption* $A(m) \in \mathcal{A}$, such that $A(m) \subseteq \mathit{States}(X)$. The availability assumption $A(m)$ states the expectation on the environment that m is invoked only when the interface state is in $A(m)$.
– For every method $m \in M$, a *call assumption* $B(m) \in \mathcal{B}$, such that $B(m) \subseteq M^L$. As for stateless SM interfaces, the call assumption $B(m)$ states the expectation on the environment that no method in $B(m)$ is invoked as a (direct or indirect) result of invoking m.

For the stateful SM interface F, we define the corresponding stateless SM interface $I_F = (M^L, M^E, \mathcal{C}, \mathcal{B})$, where $m \in C(l)$ iff some edge of the interface program $P(l)$ is labeled with a call of method m.

Pushdown game semantics. Our aim in modeling the behaviour of a software module is to capture its possible interactions with the environment (i.e., the implementations of the external methods). While the environment cannot directly change the interface state, it can do so by calling a local method. Our notion of an error is a call of a local method m when the interface state is not in $A(m)$; such a call violates the availability assumption. For every local method m and interface state $s \in A(m)$, we want to check that there is a "helpful" environment, so that along every behavior of the interface that can result from calling m in state s, no error occurs. That is, we ask if there is a way of resolving the nondeterministic choices of the environment such that for all nondeterministic choices of the interface, an error never occurs. The environment has the choice, when an external method is called, to call back any sequence of local methods; the interface has the choice, when executing a local method l, to pursue any path in the nondeterministic interface program $P(l)$. As mutually recursive procedures are naturally modeled as pushdown systems [10], we need pushdown games, with the two players Interface and Environment. Given a stateful SM interface $F = (X, M^L, M^E, \mathcal{P}, \mathcal{A}, \mathcal{B})$, a local method $m_0 \in M^L$, and an interface state $s_0 \in S$, we define the following pushdown game $\mathcal{G}(F, m_0, s_0) = (Q, \Sigma, \sigma, \Gamma, c_0, \hookrightarrow)$:

- First, for every external method $m \in M^E$ with call assumption $B(m)$, we define the interface program with two locations: the initial location $\vdash_m$ and the terminal location $\dashv_m$, an edge $(\vdash_m, \dashv_m)$ labeled by a vacuously true condition, and an edge $(\vdash_m, \vdash_m)$ for each local method $m \in M^L \setminus B(m)$, labeled by the call of method m. In other words, we allow player Environment to choose an arbitrary implementation of m by making any sequence of local method calls that are permitted by the call assumption $B(m)$.
- Let $\mathcal{L}$ be the disjoint union of the sets of program locations of $P(m)$ for all methods $m \in M$, where $M = M^L \cup M^E$. We write $\vdash_m$ for the initial location of $P(m)$, and we write $\dashv_m$ for the terminal location of $P(m)$. The set of control states of the pushdown game is $Q = \mathit{States}(X) \times \mathcal{L}$, and the stack alphabet is $\Gamma = \mathcal{L} \cup \{\bot\}$. The control state (s, ℓ) is existential (i.e., player Environment moves) if ℓ is a nonterminal location of $P(m)$ for some external method $m \in M^E$, and (s, ℓ) is universal (i.e., player Interface moves) otherwise. The initial configuration is $c_0 = ((s_0, \vdash_{m_0}), \bot)$; that is, the stack contains only the bottom marker $\bot$.
- The stack records the return locations for a sequence of method calls. The transition relation $\hookrightarrow$ of the pushdown game is defined by the following rules.
  - *Method call.* For every state $s \in \mathit{States}(X)$, edge (ℓ, ℓ') labeled with a call of method m, and stack symbol $\gamma \in \Gamma$, we have $((s, \ell), \gamma) \hookrightarrow ((s, \vdash_m), \mathit{push}(\ell'))$ iff $s \in A(m)$.
  - *Return from method call.* For every state $s \in \mathit{States}(X)$, method $m \in M$, and location $\ell \in \mathcal{L}$, we have $((s, \dashv_m), \ell) \hookrightarrow ((s, \ell), \mathit{pop})$.
  - *Conditional.* For every state $s \in \mathit{States}(X)$, edge (ℓ, ℓ') labeled with a condition c over $\mathit{States}(X)$, and stack symbol $\gamma \in \Gamma$, we have $((s, \ell), \gamma) \hookrightarrow ((s, \ell'), \mathit{skip})$ if the state s satisfies the condition c.
  - *Assignment.* For every state $s \in \mathit{States}(X)$, edge (ℓ, ℓ') labeled with an assignment $x := d$ and a stack symbol $\gamma \in \Gamma$, we have $((s, \ell), \gamma) \hookrightarrow ((s[d/x], \ell'), \mathit{skip})$.

A *play* of the pushdown game $\mathcal{G}(F, m_0, s_0)$ is a maximal sequence of configurations $c_0 \hookrightarrow c_1 \hookrightarrow c_2 \hookrightarrow \cdots$ starting from the initial configuration c_0. Player Environment *wins* the play if it is finite, and the last configuration is $((s, \dashv_{m_0}), \bot)$, for some state $s \in \mathit{States}(X)$; otherwise, player Interface is the winner. Note that if a finite play is won by player Interface, then the last configuration exhibits an error: it corresponds to a call of a local method in a state in which the method is not available. Note also that player Environment has a winning strategy in the game $\mathcal{G}(F, m_0, s_0)$ if it can interact with the interface F in such a way that if method m_0 is called in state s_0, then it can run to termination without ever violating an availability assumption. We say that (F, m_0, s_0) is *safe* if player Environment has a winning strategy in the pushdown game $\mathcal{G}(F, m_0, s_0)$.

Availability consistency and strengthening. A stateful SM interface $F = (S, M^L, M^E, \mathcal{P}, \mathcal{A}, \mathcal{B})$ is *availability consistent* if (F, m, s) is safe for all local methods $m \in M^L$ and states $s \in A(m)$. Note that the notion of availability consistency is independent of the call assumptions $\mathcal{B}$, which are not modeled by the game semantics defined above. For every stateful SM interface F,

it can be shown that there is a unique most general stateful SM interface $\mathit{Safe}(F) = (S, M^L, M^E, \mathcal{P}, \mathcal{A}', \mathcal{B})$ which is availability consistent, and obtained from F by strengthening the availability assumptions; namely, $A'(m) = \{s \in S \mid (F, m, s) \text{ is safe}\}$ for all local methods $m \in M^L$. From Theorem 2 it follows that the availability consistency of a stateful SM interface F can be checked, and $\mathit{Safe}(F)$ constructed, in exponential time.

Composition. Two stateful SM interfaces $F = (X_F, M_F^L, M_F^E, \mathcal{P}_F, \mathcal{A}_F, \mathcal{B}_F)$ and $G = (X_G, M_G^L, M_G^E, \mathcal{P}_G, \mathcal{A}_G, \mathcal{B}_G)$ are *proto-compatible* if $M_F^L \cap M_G^L = \emptyset$. If F and G are proto-compatible, then the *proto-composition* $F \oplus G = (X, M^L, M^E, \mathcal{P}, \mathcal{A}, \mathcal{B})$ is defined as follows:

- $X = X_F \uplus X_G$.
- $M^L = M_F^L \cup M_G^L$ and $M^E = (M_F^E \cup M_G^E) \setminus M^L$.
- For all local methods $l \in M^L$, we have $P(m) = P_F(m)$ if $m \in M_F^L$, and $P(m) = P_G(m)$ if $m \in M_G^L$.
- For all local methods $m \in M^L$, we have $A(m) = A_F(m) \times \mathit{States}(X_G)$ if $m \in M_F^L$, and $A(m) = \mathit{States}(X_F) \times A_G(m)$ if $m \in M_G^L$.
- For all methods $m \in M$, we have $B(m) = B_F(m) \cup B_G(m)$.

The proto-compositon $F \oplus G$ may violate call assumptions as well as availability assumptions. We first remove all violations of call assumptions by making unavailable all local methods whose call assumptions are not satisfied, and then we remove all violations of availability assumptions by strengthening. Define $F \oplus^e G = (S, M^L, M^E, \mathcal{P}, \mathcal{A}^e, \mathcal{B})$, where for each local method $l \in M^L$, we have $A^e(l) = A(l)$ if the call assumption $B(l)$ is satisfied in the stateless SM interface $I_{F \oplus G}$ (which is the same as $I_F || I_G$), and otherwise $A^e(l) = \emptyset$. The *composition* of two proto-compatible SM interfaces F and G is $F||G = \mathit{Safe}(F \oplus^e G)$.

Proposition 1. *The composition of stateful SM interfaces is associative.*

Theorem 3. *Computing the composition of two stateful SM interfaces is in DEXPTIME.*

As in the stateless case, one might define a notion of compatibility for stateful SM interfaces, but a meaningful definition often depends on the application scenario. For example, one might say that two stateful SM interfaces F and G are *compatible* if they are proto-compatible, and the composition $F||G$ has a local method m with $A_{F||G}(m) \neq \emptyset$; that is, at least one method is available in at least one state of the composition. In practice, as in the TinyOS case study below, one often adopts a stronger notion of compatibility, which requires that certain methods must be available in certain states.

An alternative definition of composition. If the call assumption $B(l)$ of a local method l is violated in the above definition of composition, then l is made unavailable in all states, that is, $A(l)$ is set to the empty set. However, if a call of method $m \in B(l)$ occurs in the interface program $P(l)$, it does not necessarily mean that m is going to be invoked when l is called; whether or not this happens may depend on the state in which l is called. This suggests the following way of relaxing the definition of composition for stateful SM interfaces.

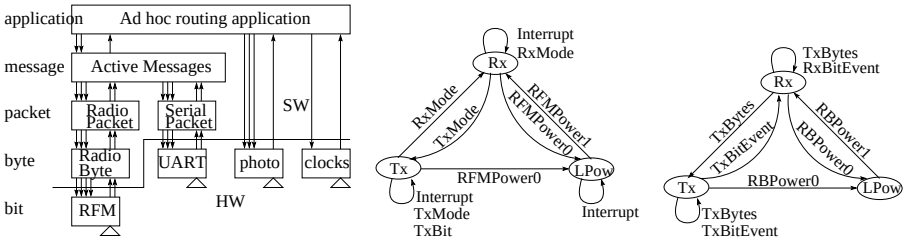
We modify the pushdown game semantics in such a way that a configuration of the pushdown game (i.e., control state and stack contents) is an error configuration if it violates a call assumption, i.e., if the method called in this configuration must not be called by a method that occurs on the stack. Note that by collecting all methods that occur on the stack in the control state of the pushdown game, we can define an error configuration by referring only to the control state. Let $\mathcal{G}^s(F \oplus G, m, t)$ be the modified pushdown game for local method m and state t of the proto-composition $F \oplus G$, and define $A^s(m) = \{t \in S_{F\oplus G} \mid \text{player Environment has a winning strategy in } \mathcal{G}^s(F \oplus G, m, t)\}$ for each local method $m \in M^L_{F\oplus G}$. The alternative composition is defined as $F||^s G = (S_{F\oplus G}, M^L_{F\oplus G}, M^E_{F\oplus G}, \mathcal{P}_{F\oplus G}, \mathcal{A}^s, \mathcal{B}_{F\oplus G})$. Note that $F||^s G$ is harder to compute than $F||G$, because the set of control states of the modified pushdown game $\mathcal{G}^s(F \oplus G, m, t)$ can be exponentially larger than that of the original game $\mathcal{G}(F \oplus G, m, t)$; this is why we chose our original definition of composition. On the other hand, the alternative composition is "better" (i.e., less constraining) in the following sense.

Proposition 2. *If a local method is available at a state t in $F||G$, then it is available at t in $F||^s G$. Moreover, for all stateful interfaces H proto-compatible with $F||G$, if a local method is available at a state t in $(F||G)||H$, then it is available at t in $(F||^s G)||^s H$.*

Example 1. Consider the following two stateful SM interfaces F and G. Interface F has two states, $\{1, 2\}$; two local methods, `a` and `b`; the program for method `a` is empty; the program for method `b` makes the deterministic choice that if in state 1, it calls method `a`, and if in state 2, it calls an external method `x`; the call assumption for method `b` is that it must not (indirectly) call itself; and both methods `a` and `b` are available in both states. Interface G has two states, $\{1, 2\}$; a local method `x`; the program for `x` makes the deterministic choice that if in state 1, it changes the state to 2 and then calls `b`, and if in state 1, it calls `a`; there are no call assumptions; and method `x` is available in both states. In the composition $F||G$ method `b` is not available in any state, because its call assumption is not satisfied in the stateless interface $I_{F\oplus G}$, and method `x` is available only in states $(1, 2)$ and $(2, 2)$, because in the other states it calls method `b` and thus violates the availability assumption for `b`. In the composition $F||^s G$, method `b` is unavailable only in state $(2, 1)$ (only when called in this state does it violate its call assumption), and method `x` is available in all states. □

5 Case Study: TinyOS

The Tiny Microthreading Operating System (TinyOS) [13] is an event-driven operating system for networked embedded sensors. The design of TinyOS uses a state-machine programming model. It consists of a scheduler and a fixed number of finite-memory modules that are arranged in layers and communicate with each other via *events* and *commands*, which cause state transitions in the modules. Events are initiated at the lowest layer by hardware interrupts. Each event may cause higher-layer events and invoke lower-layer commands, but the TinyOS



(a) TinyOS stack. (b) `RFM`. (c) `RadioByte`.

Fig. 1. 1(a) TinyOS communication stack for adhoc networking. 1(b) State transitions for `RFM` interface. 1(c) State transitions for `RadioByte` interface.

design requires that commands cannot cause events. A typical application is shown in Figure 1(a), consisting of a low-power radio stack, a UART serial port stack, sensor stacks, and adhoc routing.

We have used stateful SM interfaces to model the interfaces of two of the modules of TinyOS version 4.3, namely, `RFM` and `RadioByte`. Both commands and events are modeled as method calls; the local storage is modeled using state variables. The availability assumptions were obtained from the designers of TinyOS; the call assumptions are immediate from the TinyOS call conventions. The actual implementation of the modeled modules comprises about 460 lines of C code, with eight variables of type byte. Instead, each interface has only one variable, which can take three values. The modules `RFM` and `RadioByte` are the lowest two layers of the TinyOS stack. They both have three operating modes (states): transmitting (`Tx`), receiving (`Rx`), and low-power (`LowPow`). The state transitions are shown in Figure 1(b) and Figure 1(c).(Note, however, that the figures have no formal meaning and are only intended to help with the intuitive understanding of the TinyOS interfaces.) Invoking a local method can start a command-event chain. For example, `Interrupt` may be invoked due to a hardware interrupt when `RFM` is in states `Tx` or `Rx`, indicating that a bit has been transmitted or received, and triggering the event `TxBitEvent` or `RxBitEvent`, respectively. A detailed description of the two interfaces can be found in a technical report that accompanies this paper.

We have implemented a tool for checking the compatibility of stateful SM interfaces in Java (JDK 1.3). Composition of the two TinyOS interfaces gives a pushdown game with 117 control states and 5 stack symbols, which induces a state space of the size $5 \cdot 117 \cdot 2^{117}$. Our tool runs for about 30 minutes of CPU time on a Sun workstation with 256 MB RAM and two 200 MHz UltraSPARC CPUs. It reports an interface incompatibility at the composite state `(LowPow, LowPow)`; that is, when both modules are in low-power mode. The incompatibility is that when the module `RadioByte` is in `LowPow` mode, invoking the only available method `RBPower1` would in turn invoke `RxMode`. However, the method `RxMode`

is not available when `RFM` is in `LowPow` mode. This incompatibility has been removed from the design in later distributions.

6 Implementation

We adapt and optimize Walukiewicz's algorithm for solving parity games on pushdown systems [18] to the special case of reachability pushdown games. Given a pushdown game $\mathcal{G}$, Walukiewicz constructs a finite game $\mathcal{F}$ of size exponential in the size of the pushdown system such that winning strategies in the finite game correspond to winning strategies in the pushdown game, and vice versa. There is a state of $\mathcal{F}$ for every triple consisting of a state of $\mathcal{G}$, a stack symbol of $\mathcal{G}$, and a state set of $\mathcal{G}$. Player 1 has a winning strategy from such a state (s, γ, T) in $\mathcal{F}$ iff he has a winning strategy to either win from s in $\mathcal{G}$ with the symbol γ on the top of the stack while never popping this copy of γ from the stack, or to enter a state in the set T while popping the symbol γ from the top of the stack. If player 1 has a winning strategy from (s, γ, T) in $\mathcal{F}$, then we say that T is a target set for s and γ. Player 1 has a winning strategy from $(s, \bot)$ in $\mathcal{G}$ iff he has a winning strategy from $(s, \bot, \emptyset)$ in $\mathcal{F}$.

We avoid the explicit construction of $\mathcal{F}$ by developing a symbolic algorithm for computing the solutions of the game. Let $\{1, 2, \ldots, n\}$ be the states of the pushdown system $\mathcal{G}$. For every state s and stack symbol γ, we construct a BDD $\mathcal{B}_{s,\gamma}(\overline{x})$, where $\overline{x} = x_1, \ldots, x_n$, to represent the set of target sets for s and γ. The function *Apply* gives for every pushdown rule of $\mathcal{G}$ a BDD over the variables $\overline{x}$:

$$Apply(r) = \begin{cases} x_{s'} & \text{if } r\text{: } (s,\gamma) \hookrightarrow (s', pop), \\ \mathcal{B}_{s',\gamma} & \text{if } r\text{: } (s,\gamma) \hookrightarrow (s', skip), \\ \exists \overline{z}.\big(\mathcal{B}_{s',\gamma'}[\overline{z}/\overline{x}] \wedge \bigwedge_{i=1}^{n}(z_i \Rightarrow \mathcal{B}_{i,\gamma})\big) & \text{if } r\text{: } (s,\gamma) \hookrightarrow (s', push(\gamma')). \end{cases}$$

Our algorithm initializes all BDDs $\mathcal{B}_{s,\gamma}$ to `false` and then keeps updating their values by applying the pushdown rules in the following way:

$$\mathcal{B}_{s,\gamma} := \begin{cases} \mathcal{B}_{s,\gamma} \;\vee\; \bigvee_{r \text{ is a rule for } (s,\gamma)} Apply(r) & \text{if } s \text{ is existential,} \\ \mathcal{B}_{s,\gamma} \;\vee\; \bigwedge_{r \text{ is a rule for } (s,\gamma)} Apply(r) & \text{if } s \text{ is universal.} \end{cases}$$

We briefly discuss the *pop*, *skip*, and *push* rules when s is an existential state; the universal case is similar. A rule $(s, \gamma) \hookrightarrow (s', pop)$ allows player 1 to instantly win by popping the top symbol γ from the stack while reaching state s'; every set containing s' is then a target set for s and γ. If there is a rule $(s, \gamma) \hookrightarrow (s', skip)$, then player 1 can apply it without any change to the stack contents, and so every target set for s' and γ is also a target set for s and γ. If there is a rule $(s, \gamma) \hookrightarrow (s', push(\gamma'))$, then T is a target set for s and γ if there is a set U such that U is a target set for s' and γ', and T is already known to be a target set for u and γ, for all $u \in U$.

The symbolic implementation of Walukiewicz's algorithm has several advantages over an enumerative solution. First, it can give substantial space savings due to the compact representation of target sets by BDDs. Second, the running

time can be significantly reduced by avoiding the explicit manipulation of large state sets, as in the "saturation" algorithm of [4]. Third, the above simple BDD expressions allow for a straightforward and succinct implementation of a reachability pushdown games solver using any BDD package. In our implementation, we use the CUDD package [19] with Java wrappers provided by JMocha [2].

References

1. S. Abramsky. *Semantics of Interaction*. Lecture Notes, Oxford University, 2002.
2. R. Alur et al. jMocha: A model-checking tool that exploits design structure. *Proc. Int. Conf. Software Engineering*, pp. 835–836. IEEE, 2001.
3. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. Software Engineering & Methodology*, 6:213–249, 1997.
4. A. Boujjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. *Concurrency Theory*, LNCS 1243, pp. 135–150. Springer, 1997.
5. A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *IEEE Software*, 16:38–45, 1999.
6. L. de Alfaro and T.A. Henzinger. Interface automata. *Proc. Symp. Foundations of Software Engineering*, pp. 109–120. ACM, 2001.
7. L. de Alfaro and T.A. Henzinger. Interface theories for component-based design. *Embedded Software*, LNCS 2211, pp. 148–165. Springer, 2001.
8. R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. *Proc. Conf. Programming Language Design & Implementation*, pp. 59–69. ACM, 2001.
9. D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits*. MIT Press, 1989.
10. J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. *Computer-Aided Verification*, LNCS 2102, pp. 324–336. Springer, 2001.
11. R.B. Findler, M. Latendresse, and M. Felleisen. Behavioral contracts and behavioral subtyping. *Proc. Symp. Foundations of Software Engineering*, pp. 229–236. ACM, 2001.
12. J. Foster, M. Fändrich, and A. Aiken. A theory of type qualifiers. *Proc. Conf. Programming Language Design & Implementation*, pp. 192–203. ACM, 1999.
13. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *Proc. Conf. Architectural Support for Programming Languages & Operating Systems*, pp. 93–104. ACM, 2000.
14. J.R. Larus, S.K. Rajamani, and J. Rehof. *Behavioral Types for Structured Asynchronous Programming*. Technical Report, Microsoft Research, 2001.
15. D.C. Luckham and F. von Henke. An overview of Anna, a specification language for Ada. *IEEE Software*, 2:9–23, 1985.
16. D.L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15:330–336, 1972.
17. D.S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Software Engineering*, 21:19–31, 1995.
18. I. Walukiewicz. Pushdown processes: Games and model checking. *Information and Computation*, 164:234–263, 2001.
19. F. Somenzi. CUDD: CU decision diagram package. Technical Report, University of Colorado at Boulder, 1997.