TICC: A Tool for Interface Compatibility and Composition*

B. Thomas Adler[1], Luca de Alfaro[1], Leandro Dias Da Silva[2], Marco Faella[3], Axel Legay[1,4], Vishwanath Raman[1], and Pritam Roy[1]

[1] School of Engineering, University of California, Santa Cruz, USA
[2] EE Department, Federal University of Campina Grande, Paraiba, Brasil
[3] Dipartimento di Scienze Fisiche, Università di Napoli "Federico II", Italy
[4] Department of Computer Science, University of Liège, Belgium

Technical Report ucsc-crl-06-01
School of Engineering,
University of California, Santa Cruz.

January 27, 2006

Abstract. We present a brief overview of the tool TICC (*Tool for Interface Compatibility and Composition*). In TICC, a component interface describes both the behavior of a component, and the component's assumptions on the environment's behavior. TICC can check the compatibility of such interfaces, and analyze their emergent behavior, via a symbolic implementation of game-theoretic algorithms.

1 Overview

Open systems are systems whose behavior is jointly determined by their internal structure, and by the inputs that they receive from their environment. A component of a larger system is, therefore, an open system, as its behavior depends on the inputs it receives from the other system components. In previous work, it has been argued that *games* constitute a natural model for open systems [1, 7, 8, 5, 2]. We use games to represent the interaction between the behavior originating within a component, and the behavior originating from the component's environment. In particular, we model components as Input-Output games: the moves of Input represent the behavior the component can accept from the environment, while the moves of Output represent the behavior the component can generate.

Unlike component models based on transition systems, models based on games provide a notion of *compatibility* [7, 8, 5]. When two components P and Q are composed, we can check whether the output behavior of P satisfies the

* This research was supported in part by the NSF CAREER grant CCR-0132780, the NSF grant CCR-0234690, the ARP grants SC20050553 and SC20051123, and a F.R.I.A grant.

input requirements of Q, and vice-versa. However, we do not define P and Q to be compatible only if their input requirements are *always* satisfied. Rather, we recognize that the output behavior of P and Q can still be influenced by their residual interaction with the environment (unless the composition of P and Q is closed). Thus, we define P and Q to be compatible if there is *some* environment under which their input assumptions are mutually satisfied, and we associate with their composition $P \| Q$ the *weakest* (most general) assumptions about the environment that guarantee mutual compatibility. In game-theoretic terms, P and Q are compatible if, in their joint model, Input has a strategy to guarantee that all outputs from P to Q can be accepted by Q, and vice-versa; the environment assumption of $P \| Q$ is simply the most general such Input strategy.

These game-based component models have been called *interface theories,* and two tools for interface theories predate Ticc. The asynchronous, action-based interface theories of [7] are implemented as part of the Ptolemy toolset [10]. The tool Chic implements synchronous, variable-based interface theories closely modeled after [8, 4]. Our goal in developing Ticc was to provide an asynchronous model where components have rich communication primitives that facilitate the modeling of software and distributed systems.

In Ticc, variables encode both the local state of the components (called *modules*) and the global state of the system. Modules synchronize on actions; the occurrence of actions can cause variables to be updated. Each global variable can be updated by more than one module, so that it is both read and write-shared; restrictions ensure that variable updates are free from race-conditions. Actions in a module can appear both as input, and as output. If an action a occurs in a module P as output, but not as input, then P can generate a, but not accept it from other modules. If a occurs in P both as input and as output, then P can both generate a, and accept it from other modules. This enables the encoding of rich communication schemes, including exclusive, and many-to-many schemes, and differentiates the modules of Ticc from other modules with more restrictive communication primitives, such as I/O Automata [12] and Reactive Modules [3]. The theory behind Ticc has been presented in [6]; here, we describe the tool itself.

2 The Ticc Tool

Ticc parses interfaces, called *modules*, encoded in a guarded-command language, and builds symbolic representations for these interfaces that are used for compatibility checking and composition. Ticc is written in OCaml [11], and the symbolic algorithms rely on the MDD/BDD Glue and Cudd packages [13]. The code of Ticc is freely available and can be downloaded from *http://dvlab.cse.ucsc.edu/dvlab/Ticc.* This web site is an open Wiki that also contains the documentation for the tool, and several additional examples.

2.1 A fire detector example

We illustrate the modeling language of Ticc by means of a simple example: a fire detection system. The system is composed of a control unit and several smoke detectors. When a detector senses smoke (action *smoke*), it reports it by emitting the action *fire*. When the control unit receives action *fire* from any of the detectors, it emits the action *call_fd*, corresponding to a call to the fire department. Additionally, an input *disable* disables both the control unit and the detectors, so that the smoke sensors can be tested without triggering an alarm.

Below, we provide the code for the control unit module (`ControlUnit`), for one of the (several) fire detectors (`FireDetector1`), as well as for a faulty detecor that ignores the *disable* messages (`Faulty_FireDetector2`).

```
module ControlUnit:
    var s: [0..3] // 0=waiting, 1=alarm raised, 2=fd called, 3=disabled
    input fire:     { local: s = 0 | s = 1 ==> s' := 1
                       else  s = 2         ==>        }
    input disable:  { local: true ==> s' := 3 }
    output call_fd: { s = 1 ==> s' = 2 }
endmodule

module FireDetector1:
    var s: [0..2] // 0=idle, 1=smoke detected, 2=inactive
    input smoke1:  { local: s = 0 | s = 1 ==> s' := 1
                     else   s = 2 ==>        } // do nothing if inactive
    output fire:   { s = 1        ==> s'  = 2 }
    input fire:    { } // other modules can detect fire too
    input disable: { local: true  ==> s' := 2 }
endmodule

module Faulty_FireDetector2:
    var s: [0..2] // 0=idle, 1=smoke detected, 2=inactive
    input smoke2:  { local: s = 0 | s = 1 ==> s' := 1
                     else   s = 2 ==>        } // do nothing if inactive
    output fire:   { s = 1        ==> s'  = 2 }
    input fire:    { } // other modules can detect fire too
    // does not listen to disable action
endmodule
```

The body of each module starts with the list of its local variables; TICC supports Boolean and integral range variables. The transitions are specified using guarded commands *guard* $\Rightarrow$ *command*, where *guard* and *command* are boolean expressions over the local and global variables; as usual, primed variables refer to the values after a transition is taken. For instance, the output transition *fire* in module `FireDetector1` can be taken only when the local variable s has value 1, and it leads to a state where $s = 2$.

When the modules `ControlUnit` and `FireDetector1` are composed, they synchronize on the shared actions *fire* and *disable*. Note that action *fire* is present also as an input in `FireDetector1`, indicating that `FireDetector1` allows other modules to output *fire*. When `FireDetector1` and `ControlUnit` are composed, differently from other synchronization schemes, the action *fire* will survive in their composition *both* as an input and as an output, thus allowing `FireDetector1` $\|$ `ControlUnit` to be composed with other fire detectors.

The composition of `ControlUnit` and `Faulty_FireDetector1` goes less smoothly. When the composition receives a *disable* action, the control unit shuts down ($s = 3$), while the faulty detector remains in operation. When the faulty detector senses smoke (input *smoke*2), it will emit *fire*: if the control unit has been disabled by the *disable* action, this causes an incompatibility. TICC diagnoses this incompatibility by synthesizing the following input restrictions:

– A restriction preventing the input *disable* if the faulty detector is in state $s = 1$, that is, it has detected smoke and is about to issue *fire*.
– A restriction preventing the input *smoke2* when `ControlUnit` is at $s = 3$ (disabled).

Since the actions *disable* and *smoke2* should be acceptable at any time, the new input restrictions for these actions are a strong indication that the composition `ControlUnit` $\|$ `Faulty_FireDetector1` does not work properly.

2.2 A house repair example

As another example, we describe a model of the interaction among contractors fixing a house. The example is available from the *Ticc* wiki mentioned above. The example models a house with four rooms: K(itchen), L(iving room), B(athroom), (bed)R(oom). Each room may have undergoing repair work; this is tracked by the following global variables:

```
var K_busy, L_busy, B_busy, R_busy: bool
```

In each room, four items might need repair: plumb(ing), electr(ical), floor, and wall; their need for repair is tracked by the following global variables:

```
var K_plumb, K_electr, K_floor, K_wall: bool
```

Similar variables track the state of L(iving room), B(athroom), and (bed)R(oom). The activity state of the four contractors is tracked by the following global variables:

```
var plumb_active, electr_active, floor_active, wall_active: bool
```

A module `Breaks` models plumbing and electrical failures. A typical action of `Breaks` is as follows:

```
  output break_K_plumb : { ~K_plumb ==> K_plumb' & K_floor' & K_wall' }
```

This action models the fact that, when the plumbing in the kitchen is not broken (means “not”, and `K_plumb` tracks whether the K(itchen) plumbing works), then it can break, generating the output action `break_K_plumb`, and signaling that the kitchen plumbing, floor, and walls need repair. A module `Caller` calls the repairmen when needed; the plumber is called using the following statement:

```
  output call_plumb : { ~plumb_active &
         (K_plumb | L_plumb | B_plumb | R_plumb) ==> plumb_active' }
```

The plumber (and similarly, the other contractors) keep track of whether they are working (via a boolean variable `working`), as well as the room on which they are working via the local boolean variables `Kw`, `Lw`, `Bw`, `Rw`. When called, the plumber is initially not working on any room.

```
input call_plumb : {local: ~plumb_active ==> working' := false }
```

When an active plumber, not working on any room, sees that the K(itchen) is unoccupied (`K_busy`) and needs repair (`K_plumb`), the plumber starts work on the K(itchen):

```
output plumb_start_K : { plumb_active & ~working
             & K_plumb & ~K_busy ==> working' & Kw' & K_busy' }
```

(similarly for other rooms). While working on the kitchen, the plumber does not expect anybody else to work in it (this is expressed by the guard `Kw`):

```
input  electr_start_K : { local: ~Kw ==>  }      //     (*)
```

We considered two different electrician modules. A "correct" implementation, `Electrician`, checks that the kitchen is free before starting work on the kitchen:

```
output electr_start_K : { electr_active & ~working
             & K_electr & ~K_busy ==> working' & Kw' & K_busy' }
```

Note that above, the variable `Kw` is local to the electrician, and indicates whether the electrician is working on the kitchen; the equally-named variable `Kw` in (*) is instead local to the plumber. An "incorrect" implementation of the electrician, `WElectrician`, in the rush of getting things done, forgets to check whether somebody else is already at work in the kitchen:

```
output electr_start_K : { electr_active & ~working
             & K_electr           ==> working' & Kw' & K_busy' }
```

Ticc is able to detect that the composition of `Breaks`, `Caller`, `Plumber`, and `Electrician` is compatible, whereas it detects that the composition of `Breaks`, `Caller`, `Plumber`, and `WElectrician` is not. Thus, the protocol violation can be discovered before the complete system, consisting also of modules to repair floors and walls, is constructed. In fact, a simple check would have revealed the incompatibility already in the composition of `Plumber` and `Welectrician`. When composing `Plumber` and `Welectrician`, *Ticc* automatically synthesizes the assumption that (i) they are not both called to work, or (ii) no room needs to be repaired by both of them. We also note that the protocol violation is revealed *thanks to the input assumption of the correct module* `Plumber`. In the game-based approach that underlies *Ticc*, the input assumptions of correct modules constrain the protocol of modules that will be later composed into the system, preventing the composition of "rogue" modules. The verification of the correctness of interaction is simply a by-product of composition. This situation should be contrasted to the usual, non-game-based approach to modeling and verification. In the usual approach, detecting incompatibilities requires writing separate specifications of correctness, and can usually be performed only once all components are composed.

3 Using Ticc

Ticc is implemented as a set of functions that extends the capabilities of the OCaml command-line. The incompatibility mentioned in the fire-detector example of the previous section is exposed by the following series of OCaml commands:

```
# open Ticc;;
# parse "fire-detector-disable.si";;
# let controlunit = mk_sym "ControlUnit";;
# let wfire2 = mk_sym "Wrong_FireDetector2";;
# print_input_restriction (compose controlunit wfire2) "disable";;
# print_input_restriction (compose controlunit wfire2) "smoke2";;
```

The `mk_sym` function builds a symbolic representation of a module, given the module name. The last two lines print how the input actions have been restricted in the composition. Ticc provides a large set of primitives for the analysis of open systems, in addition to the ones illustrated above, including verification and simulation capabilities. We are considering developing a real-time extension of the tool, based on the *Timed Interfaces* of [9]. This is a large and complex endeavor, as the game-theoretic machinery of Ticc will have to be replaced with one suited to real-time games.

References

1. S. Abramsky. Semantics of interaction. In *Trees in Algebra and Programming – CAAP'96*, *LNCS* 1059, Springer-Verlag, 1996.
2. S. Abramsky, D. Ghica, A. Murawski, and L. Ong. Applying game semantics to compositional software modeling and verification. In *Proceedings of TACAS 04*, LNCS, Springer-Verlag, 2004.
3. R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.
4. A. Chakrabarti, L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. Synchronous and bidirectional component interfaces. In *Proceedings of CAV 02*, *LNCS* 2404. Springer-Verlag, 2002.
5. L. de Alfaro. Game models for open systems. In *Proceedings of the International Symposium on Verification (Theory in Practice)*, *LNCS* 2772, Springer-Verlag, 2003.
6. L. de Alfaro, L. Dias da Silva, M. Faella, A. Legay, P. Roy, and M. Sorea. Sociable interfaces. In *Procedings of FROCOS 05*, *LNAI* 3717. Springer-Verlag, 2005.
7. 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)*. ACM Press, 2001.
8. L. de Alfaro and T.A. Henzinger. Interface theories for component-based design. In *Proceedings of EMSOFT 01*, *LNCS* 2211. Springer-Verlag, 2001.
9. L. de Alfaro, T.A. Henzinger, and M. Stoelinga. Timed interfaces. In *Proceedings of EMSOFT 02*, *LNCS* 2491. Springer-Verlag, 2002.
10. E. A. Lee and Y. Xiong. A behavioral type system and its application in Ptolemy II. *Formal Aspect of Computing Journal*, 2003.
11. Xavier Leroy. Objective caml. http://www.ocaml.org.
12. N.A. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, 1996.
13. Fabio Somenzi. Cudd: Cu decision diagram package. http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html.