

Automata for Specifying Component Interfaces

Thomas A. Henzinger

University of California, Berkeley

Interfaces play a central role in component-based design and verification. An *interface* should specify no more and no less than the information necessary for checking if the corresponding component fits together with other components. This makes an interface specification different from a component specification. While component specifications are pessimistic (they must be satisfied in every environment), interface specifications are optimistic (they must be satisfiable by some environment). For example, assuming inputs x and y and output z , the formula “ $y \neq 0 \Rightarrow z = x/y$ ” is a component specification; the type declaration “ $x : \text{int}; y : \text{int} \setminus \{0\}; z : \text{rat};$ ” is an interface specification. While the former has no control over the input y , the latter puts a design constraint on y , namely, that y be different from 0. The division component can be composed with a client component if and only if the constraint $y \neq 0$ is guaranteed by the interface of the client. Note that not every input y satisfies the constraint $y \neq 0$, but it is a meaningful input constraint because it is satisfiable.

While for component specifications C and C' , composition $C||C'$ is a (more or less) total function, for interface specifications I and I' , composition $I||I'$ is partial: the composition is defined iff I and I' are *compatible*, that is, if the input constraints of I are guaranteed by the output constraints of I' , and vice versa. For simple interfaces, which constrain input and output values, compatibility checking is type checking. We present a series of formalisms for specifying richer interfaces, which constrain input and output behavior and timing, as well as computational resources such as memory and power usage. For example, the interface of a file server with two methods A (“open file”) and B (“read file”) may require that B is not called before A is called. Such behavioral interfaces can be specified using input/output automata, and compatibility checking for automaton-based interfaces is a game-theoretic problem: the interface automata I and I' are compatible iff the environment has a way (i.e., *strategy*) to provide free inputs that prevent the composite automaton $I||I'$ from entering incompatible states. For behavioral interfaces, an incompatible state arises when the input and output constraints of I and I' do not match; for example, if the server I expects a call of A , but the client I' calls instead B . For resource interfaces, an incompatible state arises when, by putting I and I' together, the available resources are exceeded. The requirement that the environment of $I||I'$ follows a strategy that avoids incompatible states (provided such a strategy exists) is then the new input constraint of the composite interface.

An algebra of interfaces contains, in addition to parallel composition, also a pre-order for refining (or abstracting) interfaces. While the *refinement* of component specifications treats inputs and outputs covariantly, as in trace containment or

simulation, the refinement of interface specifications treats inputs and outputs contravariantly, as in subtyping: a refined version of an interface may generate fewer outputs, but it must accept at least as many inputs. This leads to alternating trace containment and alternating simulation [1] for refining automaton-based interfaces.

This talk is based on joint work that originated in collaboration with Luca de Alfaro and is described in detail in the following series of papers: [6] formalizes the notions of compatibility and refinement between interfaces; [5] introduces input/output automata for specifying behavioral interfaces; [2] presents push-down interfaces for software components, and [3] defines synchronous interfaces for hardware components; [7] enhances interfaces with real-time constraints, and [4] adds resource constraints to interfaces. All of these interface formalisms are variations on the common theme of defining interfaces as games between an input player and an output player.

References

1. R. Alur, T.A. Henzinger, O. Kupferman, and M.Y. Vardi. Alternating refinement relations. In *CONCUR 98: Concurrency Theory*, Lecture Notes in Computer Science 1466, pages 163–178. Springer-Verlag, 1998.
2. A. Chakrabarti, L. de Alfaro, T.A. Henzinger, M. Jurdziński, and F.Y.C. Mang. Interface compatibility checking for software modules. In *CAV 02: Computer-Aided Verification*, Lecture Notes in Computer Science 2404, pages 428–441. Springer-Verlag, 2002.
3. A. Chakrabarti, L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. Synchronous and bidirectional component interfaces. In *CAV 02: Computer-Aided Verification*, Lecture Notes in Computer Science 2404, pages 414–427. Springer-Verlag, 2002.
4. A. Chakrabarti, L. de Alfaro, T.A. Henzinger, and M. Stoelinga. Resource interfaces. Submitted, 2003.
5. L. de Alfaro and T.A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*, pages 109–120. ACM Press, 2001.
6. L. de Alfaro and T.A. Henzinger. Interface theories for component-based design. In *EMSOFT 01: Embedded Software*, Lecture Notes in Computer Science 2211, pages 148–165. Springer-Verlag, 2001.
7. L. de Alfaro, T.A. Henzinger, and M. Stoelinga. Timed interfaces. In *EMSOFT 02: Embedded Software*, Lecture Notes in Computer Science 2491, pages 108–122. Springer-Verlag, 2002.