Interaction systems: a replacement for automata?

Extended abstract

Pierre Hyvernat

Abstract: the notion of *automaton* is familiar to anybody having interest in computer science. The notion of *interaction system* is, on the other hand, rather new. One way to see those is as a refinement of automata by distinguishing two *actors*.

Many notions from automata theory can be defined in this new context in a simple yet subtler way.

The aim of those few pages is to try to convince you that interaction systems are a better model than plain automata whenever the processes at stake are *dialog-like*...

-1: introduction

Automata and language play a fundamental role in many areas of computer science: let it be parsing, lexical analysis, model checking, specification etc.

A transition system (simple kind of automata) is usually defined as consisting of a set of vertices S (with a special initial vertex s_0) with, from any state, a number of labeled arcs to some other states. Usually, there is also a notion of *accepting states*...

The intuition used in the sequel is that transition system describe the possible behaviors of *processes*: labeled arcs correspond to actions.

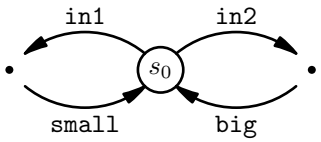

Fig 1: a chocolate vending machine

▷ **Remark:** the above figures models a simple chocolate vending machine: in the initial state, the customer can either put 1 or 2 pennies in the machine, and he will receive either a small chocolate (1 penny) or a big chocolate (2 pennies).

In real-life, almost no process is static, but most of them interacts with their environment (where the environment is represented by another process).

One special case is that of *dialog-like* systems: there are only two processes interacting with each other. The typical is that of a server and a client. The client interacts with the server (a dialog), and when the client finishes, the server is ready to interact with a new client.

Interaction systems are an attempt to make reasoning about dialog-like system built into the theory. The different ways to model dialogs in classical automata theory are not entirely satisfying (transducers, Mealy machines, input-output automata, CCS/CSP).

0: type theory and notation

All of what follows can be formalized inside Martin-Löf type theory*. No advanced notion about type theory is really needed to understand it, but it can help. I only give here the intuition to make it readable... See [ML84] or [NPS] for details.

—▷ `Set` is the collection of all "sets". The difference with classical sets is that for any S in `Set`, we know how to construct the elements of S.

—▷ If S is such a set, $\texttt{Fam}(S)$ represent the collection of families of elements of S. Such a family is given by an index set I in `Set`, and a function $f : I \to S$. This is a way to represent subsets of S: $U = \{f(i) \mid i \in I\}$.

—▷ A second way to represent subsets of S, is by giving a propositional function ϕ depending on an element of S: $U = \{s \mid \phi(s)\}$. The collection of all such propositional functions over S is called $\texttt{Pow}(S)$, and such a ϕ is called a subset of S.

* modulo the problem of dealing with co-inductive definitions.

Those two notions of subsets are identical in classical set theory, but not quite in Martin-Löf type theory.

For an obscure reason called *predicativity*, neither $\texttt{Fam}(S)$ nor $\texttt{Pow}(S)$ are in `Set`. In some sense, they are too big...

If S is a set, $s \in S$ means that s is an element of S, and if ϕ is a subset of S (written $\phi \subseteq S$), then $s \; \varepsilon \; \phi$ means that s is an element of S ($s \in S$) and that $\phi(s)$ is true.

$\multimap\triangleright$ Quantifiers $(\exists s \in S) \ldots$ or $(\forall s \in S) \ldots$ have their usual mathematical meaning and are taken constructively.

1: interaction systems

Definitions

- **Definition:** let S be a set; an interaction system on S is given by:
  - $A(s) : \texttt{Set}$ for $s \in S$;
  - $D(s,a) : \texttt{Set}$ for $s \in S$ and $a \in A(s)$;
  - $n(s,a,d) \in S$ for $s \in S$, $a \in A(s)$ and $d \in D(s,a)$;
  - $s_0 \in S$.

Let's go through this definition slowly:

- S is the set of *states;*
- $s_0 \in S$ is the *initial* state
- when in state s, the Angel (first player) has a set of possible *actions*: $A(s)$;
- after such an action, the Demon (second player) has a whole set of allowed *reactions*: $D(s,a)$;
- after a basic interaction a/d, then the state changes to $n(s,a,d)$.

Note that the Angel and the Demon play almost symmetric roles. It is usually a matter of point of view. (And from the Demon's view, it is the Angel who is "evil"...) The only difference is that the Angel gets to start the interaction.

> ▷ **Remark:** there is a more concise way to express the above definition: an interaction system on S is equivalent to a pair (f, s_0) with $f : S \to \texttt{Fam}^2(S)$ and $s_0 \in S$.
> This is less intuitive but more algebraic...

Note also that this is a deterministic notion of automata: there is no "ε-transition", and from a given state (or Demon's state), all transitions have different labels...

In the following figure, we represent the interaction system as a tree. The leaves actually refer to non-terminal node above and should be identified with those. This is to make it easier to read.

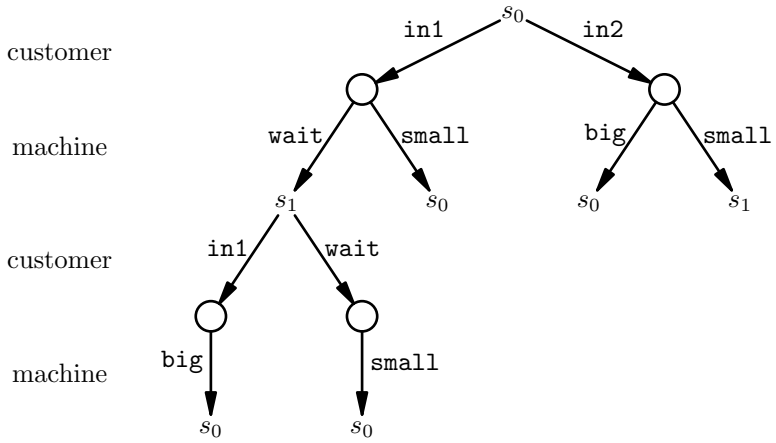

Fig 2: vending machine and interaction systems

▷ **Remark:** In an interaction system, only the Angel has a specific set of states. The "states" for the Demon consist of a pair of an Angel state and an Angel action. This is the reason for the asymmetry of the definitions.
There is an alternate notion which could be useful in practice: a *symmetric interaction system* on the sets S_A and S_D consists of:
- an initial state for the Angel: $s_A \in S_A$;
- an initial state for the Demon: $s_D \in S_D$;
- the Angel's specification: $f_A : S_A \to \mathtt{Fam}(S_D)$;
- the Demon's specification: $f_D : S_D \to \mathtt{Fam}(S_A)$.

This allows for fully symmetric definitions.

Interaction systems and transition graphs

The usual notion of transition graph is subtly different.

One general notion of transition graph could be:

- a set S of state (vertices);
- in each state s, a set of allowed actions $A(s)$ (labels);
- for each label, a subset $N(s, a)$ of new states (transitions).

A transition graph is deterministic when $N(s, a)$ is always either empty or a singleton set.

One difference between this definition and some more usual notion of transition graphs is that transition graphs usually have a single set Σ of labels (the alphabet). As far as the definition is concerned, this is mostly irrelevant. (Take $\Sigma = \bigcup_{s \in S} A(s)$...)

The difference between transition graphs and interaction systems is that while an interaction system is given by a function $f : S \to \mathtt{Fam}\big(\mathtt{Fam}(S)\big)$, a transition graph is given by a function $g : S \to \mathtt{Fam}\big(\mathtt{Pow}(S)\big)$.

Non-determinism is thus translated into Demon's choice. This makes it easier to talk about some aspect of non-determinism, and somehow gives a type to it: non determinism for action a in state s has type $D(s, a)$...

If some action appears deterministically ($N(s, a) = \{s'\}$) then the Demon's doesn't have much choice: he merely acknowledges seeing an action.

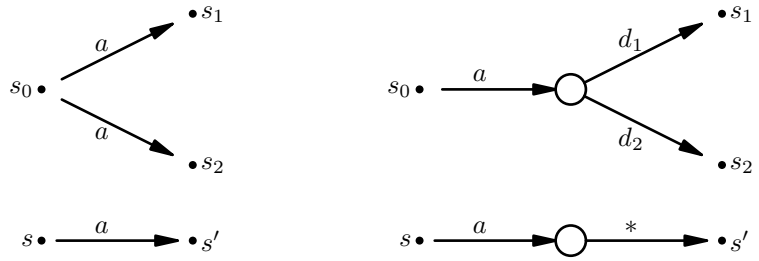

Fig 3: "Typing" non-determinism and determinism

Also, interaction systems allow to differentiate between the two processes at stake. Take a chocolate vending machine as an automata (this is slightly more complex vending machine than the one in figure 1):

```
VMC =   in2 -> big -> out1 -> VMC
      | in2 -> small -> out1 -> VMC
      | in1 -> small -> WMC
      | in1 -> in1 -> big -> VMC
```

Some problems are:

- there is no clear interleaving of action from the machine and the customer;
- there is no distinction between customer and machine.

This is little problematic: it is clear that `in2` is an action from the customer and `out1` an action from the machine, but what about `small`? Is it the customer choosing to get a small chocolate, or the machine choosing to give a small chocolate?

Compare that with figure 2. There, each action is clearly "typed" customer (Angel) or machine (Demon).

One difference is that interaction systems give a lot more importance to states.

The reason why states are more important is that they are what we can reason about. They represent the internal state of the process and are as independent of the Real World as possible.

The actions are relevant for two reasons:

- they govern the evolution of the current state;
- they have physical consequences in the Real World (increasing the value of a register, or giving a chocolate...)

The first point brings us back to the study of states, the second is more a problem of hardware construction / engineering.

Strategies

One important notion in automata theory is that of *trace*. This is a path (sequence of labels) in the labeled transition system starting from an initial state. The trace is "accepted" or "recognized" if the final state reached by the path is a final state. An accepted trace is a witness that a final state is accessible from the starting state.

What is the appropriate notion of *accessibility* in the case of interaction structure? In other words, when can the Angel know he can "reach" a final state?

The definition is inductive: the Angel can reach a final state if he already is in a final state, or if he can choose an action which will bring him "closer" to a final state, whatever the Demon does...

- **Definition:** let $F \subseteq S$, $s \in S$, we say that F is *accessible* from s (by the Angel) if:

$$s \varepsilon \mathcal{A}(F) \iff \begin{cases} s \varepsilon F \\ \text{or} \\ \big(\exists a \in A(s)\big)\ \big(\forall d \in D(s,a)\big) \quad n(s,a,d) \varepsilon \mathcal{A}(F) \end{cases}$$

  A proof that $s \varepsilon \mathcal{A}(F)$ can be seen as a (winning) strategy for the Angel to go into F...
  We also say that F *covers* s, and write $s \triangleleft F$.

The traditional notion of trace / accessibility in this context is simply:

- **Definition:** let $F \subseteq S$, $s \in S$, we say that F is *weakly accessible* from s if:

$$s \varepsilon \mathcal{W}(F) \iff \begin{cases} s \varepsilon F \\ \text{or} \\ \big(\exists a \in A(s)\big)\ \big(\exists d \in D(s,a)\big) \quad n(s,a,d) \varepsilon \mathcal{W}(F) \end{cases}$$

The difference between the two notions of accessibility is that in the first one, the Angel does not trust the Demon to play accordingly to his strategy. In the second, he actually relies on the Demon for doing half of his job*...

The Demon only exists as an opponent to the Angel. While the Angel is trying to do something (hopefully good) like reaching a final state, the Demon is doing his best to prevent him doing so...

Whenever the Angel does something, the Demon tries to answer in the worst possible way: suppose $G \subseteq S$ is a subset of "bad†" states. The Demon wants to confine the Angel to G: whatever the Angel does, he wants choose a reaction which will still restrict the Angel...

- **Definition:** let $G \subseteq S$ and $s \in S$, we say that G *restricts* s if:

$$s \varepsilon \mathcal{J}(G) \iff \begin{cases} s \varepsilon G \\ \text{and} \\ \big(\forall a \in A(s)\big)\ \big(\exists d \in D(s,a)\big) \quad n(s,a,d) \varepsilon \mathcal{J}(G) \end{cases}$$

  We write $s \ltimes G$...

---

* Something completely unheard of!

† for the Angel

Note that this is a co-inductive definition as opposed to the inductive definition of $\mathcal{A}$ or $\mathcal{W}$.

This notion is the exact dual to the notion of accessibility. There is also a notion dual to that of weak accessibility:

- ◦ **Definition:** let $G \subseteq S$ and $s \in S$, we say that G *strongly restricts* s if:

$$s \varepsilon \mathcal{S}(G) \iff \begin{cases} s \varepsilon G \\ \text{and} \\ \big(\forall a \in A(s)\big)\ \big(\forall d \in D(s,a)\big) \quad n(s,a,d) \varepsilon \mathcal{S}(G) \end{cases}$$

  This is also a co-inductive notion...

In other words, if the initial state is in G, then both the Angel and the Demon are restricted to G.

The smallest G such that $s_0 \in \mathcal{S}(G)$ is the accessible part of the interaction systems. Any state outside this G could as well be forgotten in the definition of the interaction system

Interaction / execution

Interaction occurs when both the Angel and the Demon have an idea in mind. The Angel wants to do something: $s \triangleleft U$, but the Demon tries very hard to restrict him: $s \ltimes V$.

The Angel will chose actions to go into U, and the Demon will react so that the Angel stays in V. When the Angel finally reach U (after some finite number of interactions), the Demon still knows how to restrict the Angel to V.

- ◇ **Proposition:** suppose $U, V \subseteq S$, $s \in S$, $s \triangleleft U$ and $s \ltimes V$, then there is some $s_f \varepsilon U$ (called *final state*) such that $s_f \ltimes V$.

The proof is immediate by induction on the proof that $s \triangleleft U$...

This process is called execution.

> ▷ **Remark:** This operation is similar to the $\|$ operation from CSP (see [CSP]), except that it is typed: it takes a (well-founded) Angel strategy and a (non-well-founded) Demon strategy and gives a trace of execution (plus a Demon strategy in the final state).
> $\|$ takes 2 processes and yields another process.

The typical example is when the Demon represents some kind of environment, or "server program" (an operative system) and the Angel is a user, or a (client) program running in this environment.

The user/client can probe the environment/server using actions, and the environment/server responds using reactions.

Note that since clients are well-founded, there can be no actual infinite interaction. The non-well-foundedness of the server appears in the fact that the server never runs out of responses...

2: specification

One interest of modeling processes with automata is that one can prove properties on the automata using well-understood theoretical machinery. The same is true for interaction systems...

The ideal goal is to prove that the process (program) meets its specification.

There are several approaches to this problem*:

- write the specifications and "refine" them to get a real program;
- write the specification, the program, and check/prove/test that the programs satisfies the specifications;

Using interaction systems lies somewhere between the first two points:

- write part of the specification to restrict the programs ("refinement" step), then write a specific restricted program (which automatically satisfies the first part of the specification) and check that it meets the rest of the specification.

---

* In most of the cases however, programs are checked only by testing (looking for "bugs") and no formal specification is ever written!

Liveness and safety

As was shown in [spec], any specification or property can be written as the intersection of a *safety* and a *liveness* property.

The formal definitions of safety and liveness still need to be found for interaction systems*. Here is the intuition of what they mean:

A liveness property is a property saying that "something good eventually happens." The typical example of liveness property is $s \triangleleft U$: this is a liveness property for the Angel...

A safety property says the "nothing bad ever happens." The typical example of safety property is $s \ltimes V$: this is safety property for the Demon...

Safety and interaction system

In [lamport], Leslie Lamport and Martín Abadi use *state machines* to represent safety properties. State machines can be seen as a simpler version of interaction systems, where only one of the Angel and Demon is present.

> ▷ **Remark:** Those single sided version of interaction systems are also called *transition systems*. A transition system on S is given by $s_0 \in S$ and $f : S \to \mathtt{Fam}(S)$.

Since for any interaction system $s_0 \ \varepsilon \ \mathcal{W}(S)$ (*i.e.* all accessible states are in S), interaction systems also represent safety properties: no matter what the Angel and the Demon do, they cannot escape S.

If all the states in S satisfy some "good" property, then neither the Angel nor the Demon can do anything too bad.

Example, ¿¿applications??

Let's try to devise a simple interaction system for a chocolate vending machine which can deliver small chocolates for 1 Swedish crown, and big chocolates for 2 Swedish crowns†.

The machine is definitely a server program, so that the machine actions are best represented as Demon's reactions. The Angel is then the customer trying to get his chocolate out of the machine.

> ▷ **Remark:** Another way to represent looping programs is by having them as Angel's strategies which go back to the initial state (a loop), or more generally, which goes back to a state *refining* the initial state. (See [inter] for the notion of refinement.)

—▷ Set of state

The customer (Angel) can essentially be in 3 states:

- s_0: "I haven't given any money";
- s_1: "I have given 1 penny";
- s_2: "I have given 2 pennies".

We put $S = \{s_0, s_1, s_2\}$, and the initial state is s_0.

—▷ Angel's actions

While in state s_0, the customer has 2 possible actions:

- `in1`: put 1 penny in the machine;
- `in2`: put 2 pennies in the machine.

While in state s_1, the customer has 2 actions:

- `in1`: put 1 penny;
- `small`: push the "small chocolate" button.

---

* This can probably only be done with the symmetric version of interaction systems. See remark on page 3

† You can replace crowns by Euros, small by big and big by huge if you want more chocolate!

While in state s_2, there are 2 actions available:

- `small`: push the "small chocolate" button;
- `big`: ouch the "big chocolate" button.

Thus $A(s_0) = \{\texttt{in1}, \texttt{in2}\}$, $A(s_1) = \{\texttt{in1}, \texttt{small}\}$, $A(s_2) = \{\texttt{small}, \texttt{big}\}$.

—▷ Reactions

Here is the list of reactions for the machine:

- $D(s_0, \texttt{in1}) = \{\texttt{small}, \texttt{wait}, \texttt{out1}\}$;
- $D(s_0, \texttt{in2}) = \{\texttt{small}, \texttt{big}, \texttt{wait}, \texttt{out1}, \texttt{out2}\}$;
- $D(s_1, \texttt{in1}) = \{\texttt{small}, \texttt{big}, \texttt{wait}, \texttt{out1}, \texttt{out2}\}$;
- $D(s_1, \texttt{small}) = \{\texttt{small}, \texttt{out1}\}$;
- $D(s_2, \texttt{small}) = \{\texttt{small}, \texttt{out1}, \texttt{out2}\}$;
- $D(s_2, \texttt{big}) = \{\texttt{big}, \texttt{out1}, \texttt{out2}\}$.

where the explanation is:

- `small` and `big`: give a small/big chocolate;
- `wait`: don't give anything, and wait for another action;
- `out1` and `out2`: give back 1 or 2 pennies.

—▷ Next state function

- $n(s_0, \texttt{in1}, \texttt{small}) = s_0$;
- $n(s_0, \texttt{in1}, \texttt{wait}) = s_1$;
- the rest is left as a trivial exercice...

Note that each transition $s \to n(s, a, d)$ will have some physical consequence in the Real World: releasing a chocolate or a coin, disabling/enabling buttons etc.

Note that this interaction system is designed so as to make it impossible to lose money (for the machine). Any server program written on this machine will be "safe from the vendor point of view."

This specifies the hardware the machine will use.

This interaction system also satisfies the following safety properties:

- the customer cannot loose more than 1 pennies (if he leaves after the machines has reacted);
- there is always a way for the customer to get his money worth of goods;
- if the customer has enough time to ask for some special chocolate, his demand will be honored, or he gets his money back.

We can now write some specific programs which:

- always always asks what the customer wants (and by the above, give what he asked for);
- always gives back money (no chocolate);
- always gives small chocolate;
- always gives big chocolate;
- if given 2 pennies, give a small chocolate and if given 1 penny, wait for a second one to give a big chocolate (illogical).

All of these are software specification, but we don't need to check that they don't loose money. (Which is probably the single most important property the vendor wants to ensure about his machines.)

∞: conclusion

The theory of interaction systems seems thus worth studying for the field of formal specification and verification. One hope is that algorithms from model checking can be used in this new context to prove some properties automatically.

One aspect which is missing, is the notion of "true" non-determinism. Non determinism doesn't always arise because one actor might make choices, but also because several actor can interact independently before making a choice. Since the theory of interaction system can only talk about dialogs between two actors, this kind of non-determinism (important in practice) cannot be dealt with.

One other interest of the structure of interaction systems is that they model almost perfectly the notion of (non-distributive) *formal topology* (see [BP3]):

- (bases of) topological spaces are interaction systems;
- continuous functions are "simulations";
- there is an interactive notion of equality between continuous relations.

This field is quite distant from the Real World, but some topological results can give insights for the more down to earth notions... (See [inter] for more details.)

Bibliography