Masaccio*: A Formal Model for Embedded Components**

Thomas A. Henzinger

University of California, Berkeley
tah@eecs.berkeley.edu

Abstract. Masaccio is a formal model for hybrid dynamical systems which are built from atomic discrete components (difference equations) and atomic continuous components (differential equations) by parallel and serial composition, arbitrarily nested. Each system component consists of an interface, which determines the possible ways of using the component, and a set of executions, which define the possible behaviors of the component in real time.

We formally define a class of entities called "components." The intended use of components is to provide a formal, structured model for software and hardware that interacts with a physical environment in real time. The model is formal in that it defines a component as a mathematical object, which can be analyzed. The model is structured in that it permits the hierarchical definition of a component, and the hierarchy can be exploited for structuring the analysis. Components are built from atomic components using six operations: parallel composition, serial composition, renaming of variables (data), renaming of locations (control), hiding of variables, and hiding of locations. There are two kinds of atomic components. An atomic discrete component is a difference equation which governs the instantaneous change of state. An atomic continuous component is a differential equation, which governs the evolutionary change of state over time. The mathematical semantics of a component is given by its *interface* and its *set of executions*. The interface of a component determines how the component can interact (be composed) with other components. Each execution specifies a possible behavior of the component as a sequence of instantaneous and evolutionary state changes.

The interface of a component Data enters and exits a component through variables; control enters and exits through locations. All variables are assumed to be typed, with domains such as the booleans $\mathbb{B}$, the nonnegative integers $\mathbb{N}$, and the reals $\mathbb{R}$. For each variable x, we assume that there is a primed version x' which has the same type as x. For a set V of variables, we denote by $[V]$ the set of type-conforming value assignments to the variables in V: if $x \in V$ and $q \in [V]$, then $q(x)$ is the value assigned by q to x. The *interface of a component* A consists of five parts:

* Version 1.0 (May 2000).

** This research was supported in part by the DARPA grants NAG2-1214 and F33615-C-98-3614, and by the MARCO grant 98-DT-660.

J. van Leeuwen et al. (Eds.): IFIP TCS 2000, LNCS 1872, pp. 549–563, 2000.

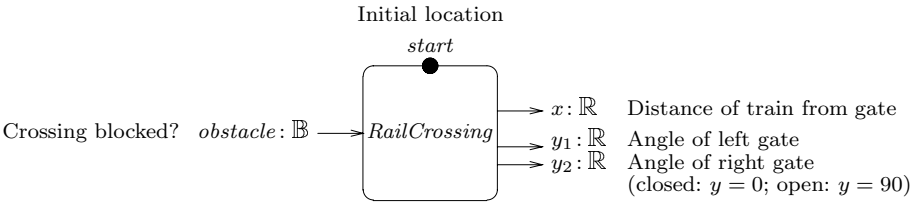

Fig. 1. A railroad crossing (safety: always $[|x| < 100 \Rightarrow y_1 = y_2 = 0]$)

- A finite set V_A^{in} of *input variables*. We write $V_A^{in'}$ for the set of primed variables whose unprimed versions are input variables.
- A finite set V_A^{out} of *output variables*. We require that the input and output variables are disjoint; that is, $V_A^{in} \cap V_A^{out} = \emptyset$. We refer to the collection $V_A^{in,out} = V_A^{in} \cup V_A^{out}$ of input and output variables as *I/O variables*. The value assignments in $[V_A^{in,out}]$ are called *I/O states*. Given an I/O state q, we denote by q' the value assignment in $[V_A^{in'}]$ which is derived from q in the following way: $q'(x') = q(x)$ for all input variables $x \in V_A^{in}$.
- A binary relation $\prec_A \subseteq V_A^{in,out} \times V_A^{out}$ of *dependencies* between I/O variables and output variables. The value of an output variable y can depend on previous values of any I/O variable x; intuitively, if $x \prec_A y$, then the value of y can depend, without delay, also on the concurrent value of x. A set U of I/O variables is *dependency-closed* if for all $x, y \in V_A^{in,out}$, if $x \prec_A y$ and $y \in U$, then $x \in U$. For example, the set V_A^{in} of input variables is dependency-closed.
- A finite set L_A^{intf} of *interface locations*. These are the locations through which control can enter or exit the component A.
- For each interface location $a \in L_A^{intf}$, a predicate $\varphi_A^{en}(a)$ on the variables in $V_A^{in,out} \cup V_A^{in'}$. Thus, given two I/O states p and q, the *entry condition* $\varphi_A^{en}(a)$ is either true or false at (p, q'), i.e., if each unprimed variable $x \in V_A^{in,out}$ is assigned the value $p(x)$, and each primed variable $y' \in V_A^{in'}$ is assigned the value $q'(y')$. Intuitively, if the current I/O state is p, and the input portion of the next I/O state is q', then the component A can be entered at location a iff the entry condition $\varphi_A^{en}(a)$ is true at (p, q').

We will distinguish between discrete and hybrid components. If A is a discrete component, then all I/O variables of A have discrete types, such as $\mathbb{B}$ or $\mathbb{N}$. Hybrid components have also I/O variables of type $\mathbb{R}$.

The executions of a component The possible finite behaviors of a component are called executions. Consider a component A. A *jump* of A is a pair $(p, q) \in [V_A^{in,out}] \times [V_A^{in,out}]$ of I/O states. The observation p is called the *source* of the jump, and q is the *sink*. A *flow* of A is a pair (δ, f) consisting of a positive real $\delta \in \mathbb{R}_{>0}$, and a function $f : \mathbb{R} \to [V_A^{in,out}]$ from the reals to I/O states which

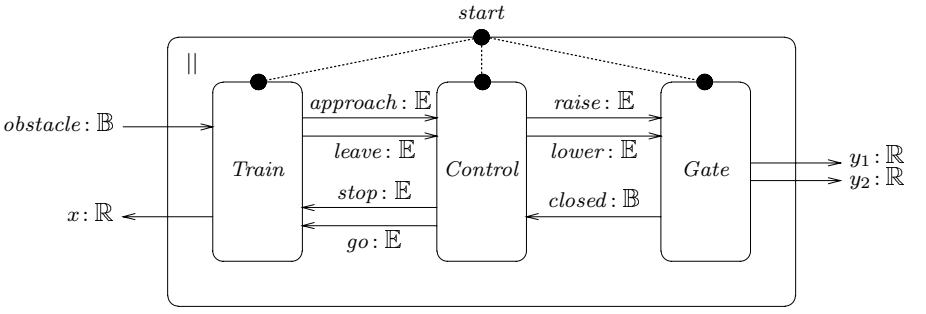

Fig. 2. The component *RailCrossing*

is differentiable[3] on the compact interval $[0, \delta] \subset \mathbb{R}$. The real δ is called the *duration* of the flow, the observation $f(0)$ is the *source*, and the observation $f(\delta)$ is the *sink*. A *step* of A is either a jump or a flow of A. The step w is *successive* to the step v if the sink of v is equal to the source of w. An *execution* of A is either a pair $(a, \boldsymbol{w})$ or a triple $(a, \boldsymbol{w}, b)$, where $a, b \in L_A^{intf}$ are interface locations and $\boldsymbol{w} = w_0 \cdots w_n$ is a nonempty, finite sequence of steps of A such that (1) the first step w_0 is a jump, and (2) each subsequent step w_i, for $1 \leq i \leq n$, is successive to the immediately preceding step w_{i-1}. The location a is called the *origin* of the execution, the sequence $\boldsymbol{w}$ is the *trace*, and the location b (when present) is the *destination*. If A is a discrete component, then all traces of A consist of jumps only; the traces of hybrid components contain also flows. We write E_A for the set of executions of the component A. We require that E_A is prefix-closed, deadlock-free, and input-permissive. Prefix closure ensures that the executions of a component can be generated operationally in a stepwise manner. The set E_A of executions is *prefix-closed* if the following four conditions are satisfied:

1. If $(a, \boldsymbol{w}, b) \in E_A$, then $(a, \boldsymbol{w}) \in E_A$.
2. If $(a, w_0 \cdots w_n) \in E_A$ for $n \geq 1$, then $(a, w_0 \cdots w_{n-1}) \in E_A$.
3. If $(a, \boldsymbol{w} \cdot (\delta, f)) \in E_A$ for a flow (δ, f), then $(a, \boldsymbol{w} \cdot (\varepsilon, f)) \in E_A$ for all reals $\varepsilon \in (0, \delta)$.
4. If $(a, (p, q)) \in E_A$ for a jump (p, q), then the entry condition $\varphi_A^{en}(a)$ is true at (p, q').

Deadlock freedom ensures that the stepwise generation of executions cannot deadlock inside a component. The set E_A of executions is *deadlock-free* if the following two conditions are satisfied:

1. For all interface locations a and I/O states p, if the entry condition $\varphi_A^{en}(a)$ is true at (p, q') for some I/O state q, then $(a, (p, q)) \in E_A$ for some jump (p, q). In other words, if the entry condition of location a is satisfiable at the I/O state p, then there is an execution with origin a and source p.

[3] On types other than $\mathbb{R}$, it can be assumed that only the constant functions are differentiable.

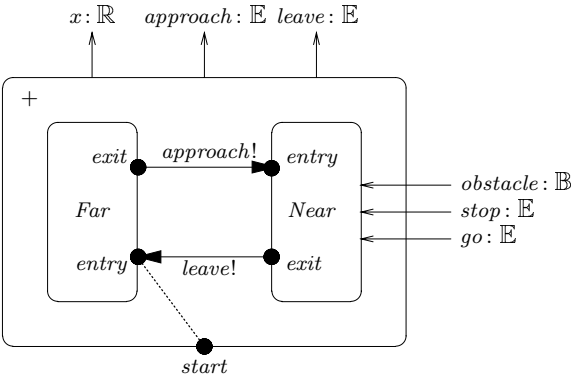

Fig. 3. The component *Train*

2. If $(a, \boldsymbol{w}) \in E_A$, then either $(a, \boldsymbol{w}, b) \in E_A$ for some interface location b, or $(a, \boldsymbol{w} \cdot (p, q)) \in E_A$ for some jump (p, q). In other words, every execution which does not have a destination can be prolonged by either a destination or a jump.

Input permissiveness ensures that a component cannot constrain the behavior of input variables. The set E_A of executions is *input-permissive* if the following two conditions are satisfied:

1. If $(a, (p, q_1)) \in E_A$ for a jump (p, q_1), then for every dependency-closed set U of I/O variables, and every I/O state q_2 such that (1) the I/O state q_2 agrees with q_1 on the variables in U and (2) the entry condition $\varphi_A^{en}(a)$ is true at (p, q_2'), there is an execution $(a, (p, q)) \in E_A$ whose sink q agrees with q_2 on the variables in U and the input variables.
2. If $(a, \boldsymbol{w} \cdot (p, q_1)) \in E_A$ for a nonempty trace $\boldsymbol{w}$ and a jump (p, q_1), then for every dependency-closed set U of I/O variables, and every I/O state q_2 which agrees with q_1 on the variables in U, there is an execution $(a, \boldsymbol{w} \cdot (p, q)) \in E_A$ whose sink q agrees with q_2 on the variables in U and the input variables.

If two components A and B have the same interface, then they can take each other's place in all contexts. We say that A *refines* (or *implements*) B if (1) the components A and B have the same interface and (2) every execution of A is also an execution of B; that is, $E_A \subseteq E_B$. If A refines B, then B can be thought of as a more abstract (permissive) version of A, with some details (constraints) left out in B which are spelt out in A. Since the executions of A are deadlock-free, if B has an execution with origin a, and A refines B, then A must also have an execution with origin a. Thus a component with a nonempty set of executions cannot be trivially implemented by a component with the empty set of executions.

The parallel composition of components Two components A and B *can be composed in parallel* if their interfaces satisfy the following three conditions:

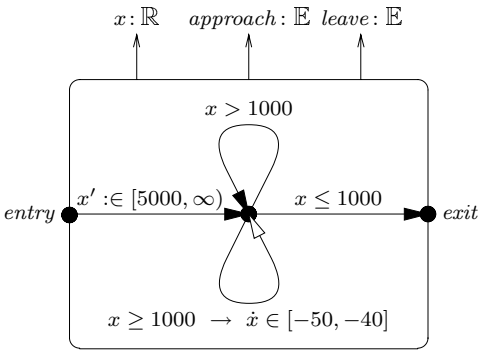

Fig. 4. The component *Far*

1. The output variables of A and B are disjoint; that is, $V_A^{out} \cap V_B^{out} = \emptyset$.
2. There is no inferred mutual dependency between an output variable of A and an output variable of B; that is, there do not exist two variables $x \in V_A^{out}$ and $y \in V_B^{out}$ such that both $x \prec_B^* y$ and $y \prec_A^* x$, where $\prec^*$ is the transitive closure of the dependency relation $\prec$.
3. For each interface location a common to both A and B, the entry conditions of a are equivalent in A and B; that is, if $a \in L_A^{intf} \cap L_B^{intf}$, then the entry condition $\varphi_A^{en}(a)$ is equivalent to the entry condition $\varphi_B^{en}(a)$. This implies, in particular, that $\varphi_A^{en}(a)$ does not constrain the primed outputs of B, nor does $\varphi_B^{en}(a)$ constrain the primed outputs of A.

If the components A and B can be composed in parallel, then $A||B$ is again a component. The *interface of the component* $A||B$ is defined from the interfaces of the subcomponents A and B:

- A variable is an input to $A||B$ if it is an input to A but not an output of B, or an input to B but not an output of A; that is, $V_{A||B}^{in} = (V_A^{in} \backslash V_B^{out}) \cup (V_B^{in} \backslash V_A^{out})$.
- A variable is an output of $A||B$ if it is an output of A or an output of B; that is, $V_{A||B}^{out} = V_A^{out} \cup V_B^{out}$.
- The dependencies of $A||B$ are inherited from both A and B; that is, $\prec_{A||B} = \prec_A \cup \prec_B$.
- The interface locations of $A||B$ are the interface locations of A together with the interface locations of B; that is, $L_{A||B}^{intf} = L_A^{intf} \cup L_B^{intf}$.
- If a is an interface location of both subcomponents A and B, then they agree on the entry condition, and this is also the entry condition of $A||B$; that is, if $a \in L_A^{intf} \cap L_B^{intf}$, then $\varphi_{A||B}^{en}(a) = (\exists V_A')\, \varphi_A^{en}(a) = (\exists V_B')\, \varphi_B^{en}(a)$, where $x' \in V_A'$ iff $x \in V_A^{in} \cap V_B^{out}$, and $y' \in V_B'$ iff $y \in V_B^{in} \cap V_A^{out}$. It follows that the component $A||B$ can be entered at location a iff both subcomponents A and B can be entered concurrently at a. The quantifiers (whose force, existential or universal, is immaterial) ensure syntactically that no primed

output variables occur freely in entry conditions. All other interface locations of $A||B$ have the unsatisfiable entry condition; that is, if $a \in L_A^{intf} \backslash L_B^{intf}$ or $a \in L_B^{intf} \backslash L_A^{intf}$, then $\varphi_{A||B}^{en}(a) = \mathit{false}$. These locations can be used only to exit the component $A||B$.

The *executions of the component* $A||B$ are defined from the executions of the subcomponents A and B:

- The pair $(a, \boldsymbol{w})$ is an execution of $A||B$ iff $(a, \boldsymbol{w}|_A)$ is an execution of A and $(a, \boldsymbol{w}|_B)$ is an execution of B, where $\boldsymbol{w}|_C$ is the restriction of the trace $\boldsymbol{w}$ to values for the I/O variables of the component C.
- The triple $(a, \boldsymbol{w}, b)$ is an execution of $A||B$ iff either $(a, \boldsymbol{w}|_A, b)$ is an execution of A and $(a, \boldsymbol{w}|_B)$ is an execution of B, or $(a, \boldsymbol{w}|_B, b)$ is an execution of B and $(a, \boldsymbol{w}|_A)$ is an execution of A.

In other words, parallel composition acts conjunctively on traces. In particular, each jump of A corresponds to a concurrent jump of B, and each flow of A corresponds to a concurrent flow of B with the same duration. If an execution of A reaches a destination, then the concurrent execution of B is terminated; if B reaches a destination, then the concurrent execution of A is terminated; if both A and B simultaneously reach destinations, then one of the two destinations is chosen nondeterministically. Note that the operator $||$ for parallel composition is associative and commutative. Furthermore, the refinement relation is preserved by parallel composition: if A and B are two components with the same interface, if A refines B, and if A (and therefore also B) can be composed in parallel with a component C, then $A||C$ refines $B||C$.

The serial composition of components Two components A and B *can be composed in series* if their interfaces agree on the output variables; that is, $V_A^{out} = V_B^{out}$. If the components A and B can be composed in series, then $A+B$ is again a component. The *interface of the component* $A+B$ is defined from the interfaces of the subcomponents A and B:

- A variable is an input to $A+B$ if it is an input to A or an input to B; that is, $V_{A+B}^{in} = V_A^{in} \cup V_B^{in}$.
- As A and B agree on their outputs, these are also the outputs of $A+B$; that is, $V_{A+B}^{out} = V_A^{out} = V_B^{out}$.
- The dependencies of $A+B$ are inherited from both A and B; that is, $\prec_{A+B} = \prec_A \cup \prec_B$.
- The interface locations of $A+B$ are the interface locations of A together with the interface locations of B; that is, $L_{A+B}^{intf} = L_A^{intf} \cup L_B^{intf}$.
- If a is an interface location of both A and B, then the entry condition of a in $A+B$ is the disjunction of the entry conditions of a in the subcomponents A and B; that is, if $a \in L_A^{intf} \cap L_B^{intf}$, then $\varphi_{A+B}^{en}(a) = \varphi_A^{en}(a) \vee \varphi_B^{en}(a)$. If a is an interface location of A but not of B, then the entry condition of a in $A+B$ is inherited from A; that is, if $a \in L_A^{intf} \backslash L_B^{intf}$, then $\varphi_{A+B}^{en}(a) = \varphi_A^{en}(a)$. If a is an interface location of B but not of A, then the entry condition of a in

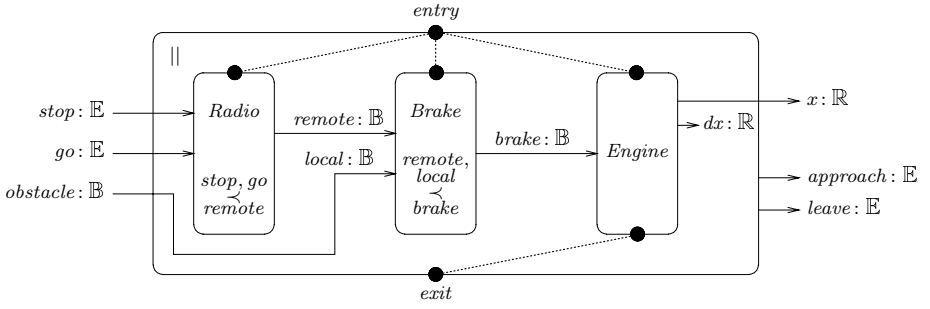

Fig. 5. The component *Near*

$A+B$ is inherited from B; that is, if $a \in L_B^{intf} \backslash L_A^{intf}$, then $\varphi_{A+B}^{en}(a) = \varphi_B^{en}(a)$. This is because the component $A + B$ is entered at location a iff either subcomponent A or subcomponent B is entered at a.

The *executions of the component* $A + B$ are defined from the executions of the subcomponents A and B:

- The pair $(a, \boldsymbol{w})$ is an execution of $A + B$ iff either $(a, \boldsymbol{w}|_A)$ is an execution of A, or $(a, \boldsymbol{w}|_B)$ is an execution of B.
- The triple $(a, \boldsymbol{w}, b)$ is an execution of $A + B$ iff either $(a, \boldsymbol{w}|_A, b)$ is an execution of A, or $(a, \boldsymbol{w}|_B, b)$ is an execution of B.

In other words, serial composition acts disjunctively on traces. Note that the operator $+$ for serial composition is associative, commutative, and idempotent. Furthermore, the refinement relation is preserved by serial composition: if A and B are two components with the same interface, if A refines B, and if A (and therefore also B) can be composed in series with a component C, then $A + C$ refines $B + C$.

Variable renaming When constructing a parallel composition $A||B$, inputs of A can be identified with outputs of B, and vice versa, by renaming variables. The variable x *can be renamed to* y in component A if x is an I/O variable of A and y is different from all I/O variables of A; that is, $x \in V_A^{in,out}$ and $y \notin V_A^{in,out}$. If x can be renamed to y in A, then $A[x := y]$ is again a component. The *interface of the component* $A[x := y]$ is defined from the interface of A: let $V_{A[x:=y]}^{in} = (V_A^{in} \backslash \{x\}) \cup \{y\}$, let $V_{A[x:=y]}^{out} = (V_A^{out} \backslash \{x\}) \cup \{y\}$, let $L_{A[x:=y]}^{intf} = L_A^{intf}$, and let $\prec_{A[x:=y]}$ and $\varphi_{A[x:=y]}^{en}$ result from renaming x to y in $\prec_A$ and in φ_A^{en}, respectively. The *executions of the component* $A[x := y]$ result from renaming x to y in the traces of the executions of A. The refinement relation is preserved by the renaming of variables: if A and B are two components with the same interface, if A refines B, and if x can be renamed to y in A (and therefore also in B), then $A[x := y]$ refines $B[x := y]$.

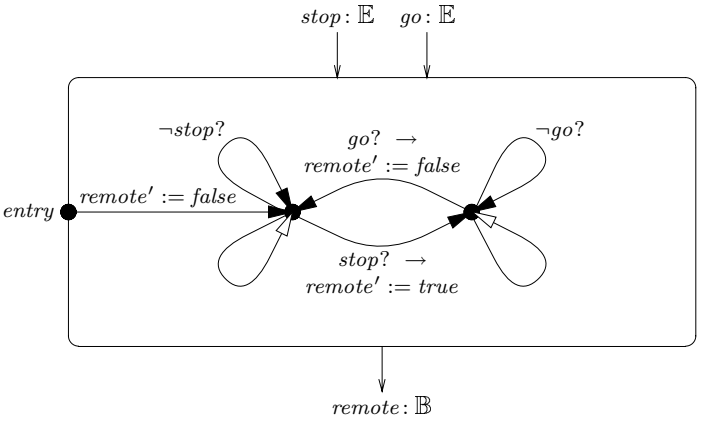

Fig. 6. The component *Radio*

Location renaming When constructing a serial composition $A + B$, interface locations of A can be identified with interface locations of B by renaming locations. The location a *can be renamed to* b in component A if a is an interface location of A; that is, $a \in L_A^{intf}$. The location b may or may not be an interface location of A. If a can be renamed to b in A, then $A[a := b]$ is again a component. The *interface of the component* $A[a := b]$ is defined from the interface of A: let $V_{A[a:=b]}^{in} = V_A^{in}$, let $V_{A[a:=b]}^{out} = V_A^{out}$, let $\prec_{A[a:=b]} = \prec_A$, let $L_{A[a:=b]}^{intf} = (L_A^{intf} \backslash \{a\}) \cup \{b\}$, let $\varphi_{A[a:=b]}^{en}(b) = \varphi_A^{en}(a)$ if $b \notin L_A^{intf}$, let $\varphi_{A[a:=b]}^{en}(b) = \varphi_A^{en}(a) \vee \varphi_A^{en}(b)$ if $b \in L_A^{intf}$, and let $\varphi_{A[a:=b]}^{en}(c) = \varphi_A^{en}(c)$ for all locations $c \in L_A^{intf} \backslash \{a, b\}$. Consequently, if both a and b are interface locations of A, then the component $A[a := b]$ can be entered at location b whenever the original component A can be entered at either a or b. The *executions of the component* $A[a := b]$ result from renaming a to b in the origins and destinations of the executions of A. The refinement relation is preserved by the renaming of locations: if A and B are two components with the same interface, if A refines B, and if a can be renamed to b in A (and therefore also in B), then $A[a := b]$ refines $B[a := b]$.

Variable hiding Hiding renders a variable local to a component, and invisible to the outside. Hidden variables do not maintain their values from one exit of a component to a subsequent entry, but they are nondeterministically reinitialized upon every entry to the component as to satisfy the applicable entry condition. The variable x *can be hidden* in the component A if x is an output variable of A; that is, $x \in V_A^{out}$. If x can be hidden in A, then $A \backslash x$ is again a component. The *interface of the component* $A \backslash x$ is defined from the interface of A: let $V_{A\backslash x}^{in} = V_A^{in}$, let $V_{A\backslash x}^{out} = V_A^{out} \backslash \{x\}$, let $\prec_{A\backslash x}$ be the intersection of the transitive closure $\prec_A^*$ with $V_{A\backslash x}^{in,out} \times V_{A\backslash x}^{out}$, let $L_{A\backslash x}^{intf} = L_A^{intf}$, and let $\varphi_{A\backslash x}^{en}(a) = (\exists x)\, \varphi_A^{en}(a)$

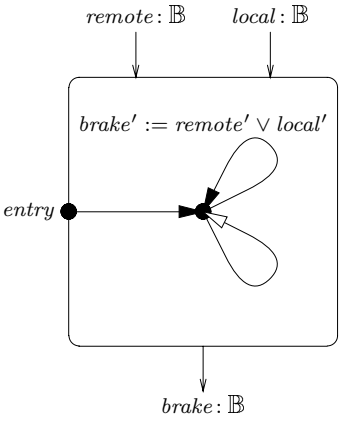

Fig. 7. The component *Brake*

for all locations $a \in L_A^{intf}$. In other words, upon entry of the component $A\backslash x$ at location a, the output variable x has an unknown value which permits the satisfaction of the entry condition $\varphi_A^{en}(a)$. The *executions of the component* $A\backslash x$ result from restricting the traces of the executions of A to values for the I/O variables of $A\backslash x$. Note that the component $A\backslash x\backslash y$ is identical to the component $A\backslash y\backslash x$. Furthermore, the refinement relation is preserved by the hiding of variables: if A and B are two components with the same interface, if A refines B, and if x can be hidden in A (and therefore also in B), then $A\backslash x$ refines $B\backslash x$.

Location hiding Hiding renders a location internal to a component, and inaccessible from the outside. The location c *can be hidden* in the component A if c is an interface location of A and the entry condition $\varphi_A^{en}(c)$ is valid; that is, $c \in L_A^{intf}$, and $\varphi_A^{en}(c)$ is equivalent to *true*. Consequently, an interface location c of A can be hidden only if the component A cannot deadlock at c, no matter what the current I/O state and the next inputs. If c can be hidden in A, then $A\backslash c$ is again a component. The *interface of the component* $A\backslash c$ is defined from the interface of A: let $V_{A\backslash c}^{in} = V_A^{in}$, let $V_{A\backslash c}^{out} = V_A^{out}$, let $\prec_{A\backslash c} = \prec_A$, let $L_{A\backslash c}^{intf} = L_A^{intf}\backslash\{c\}$, and let $\varphi_{A\backslash c}^{en}(a) = \varphi_A^{en}(a)$ for all locations $a \in L_{A\backslash c}^{intf}$. The *executions of the component* $A\backslash c$ are defined from the executions of A:

- The pair $(a, \boldsymbol{w})$ is an execution of $A\backslash c$ iff $c \neq a$ and either $(a, \boldsymbol{w})$ is an execution of A, or there is a finite sequence $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_n$ of traces, $n \geq 2$, such that $\boldsymbol{w} = \boldsymbol{w}_1 \cdots \boldsymbol{w}_n$ and the following are all executions of A: the triple $(a, \boldsymbol{w}_1, c)$, the triples $(c, \boldsymbol{w}_i, c)$ for all $1 < i < n$, and the pair $(c, \boldsymbol{w}_n)$.
- The triple $(a, \boldsymbol{w}, b)$ is an execution of $A\backslash c$ iff $c \notin \{a, b\}$ and either $(a, \boldsymbol{w}, b)$ is an execution of A, or there is a finite sequence $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_n$ of traces, $n \geq 2$, such that $\boldsymbol{w} = \boldsymbol{w}_1 \cdots \boldsymbol{w}_n$ and the following are all executions of A: the triple $(a, \boldsymbol{w}_1, c)$, the triples $(c, \boldsymbol{w}_i, c)$ for all $1 < i < n$, and the triple $(c, \boldsymbol{w}_n, b)$.

In other words, the executions of $A\backslash c$ result from stringing together, at location c,

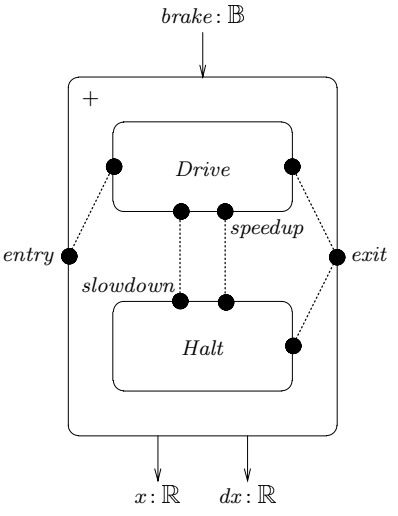

Fig. 8. The component *Engine*

a finite number of executions of A. Note that the component $A\backslash c\backslash d$ is identical to the component $A\backslash d\backslash c$. Furthermore, the refinement relation is preserved by the hiding of locations: if A and B are two components with the same interface, if A refines B, and if c can be hidden in A (and therefore also in B), then $A\backslash c$ refines $B\backslash c$.

Atomic discrete components The *discrete components* are built from atomic discrete components using the six operations of parallel and serial composition, variable and location renaming, and variable and location hiding. Each *atomic discrete component* is specified by a jump action. A *jump action* J consists of a finite set X_J of *source variables*, a finite set Y_J of *uncontrolled sink variables*, a finite set Z_J of *controlled sink variables* disjoint from Y_J, and a predicate φ_J^{jump} on the variables in $X_J \cup Y_J' \cup Z_J'$, where V' is the set of primed versions of the variables in V. The predicate φ_J^{jump} is called *jump predicate*; it is typically written as a guarded difference equation. The jump action J specifies the component $A(J)$. The *interface of the component* $A(J)$ is defined as follows:

- The inputs to $A(J)$ are the source variables of J which are not controlled sink variables, together with the uncontrolled sink variables; that is, $V_{A(J)}^{in} = (X_J \backslash Z_J) \cup Y_J$.
- The outputs of $A(J)$ are the controlled sink variables of J; that is, $V_{A(J)}^{out} = Z_J$.
- Each controlled sink variable depends on each uncontrolled sink variable; that is, for all $x \in V_{A(J)}^{in,out}$ and $y \in V_{A(J)}^{out}$, define $x \prec_{A(J)} y$ iff $x \in Y_J$ and $y \in Z_J$.
- The component $A(J)$ has two interface locations, say, *from* and *to*; that is,

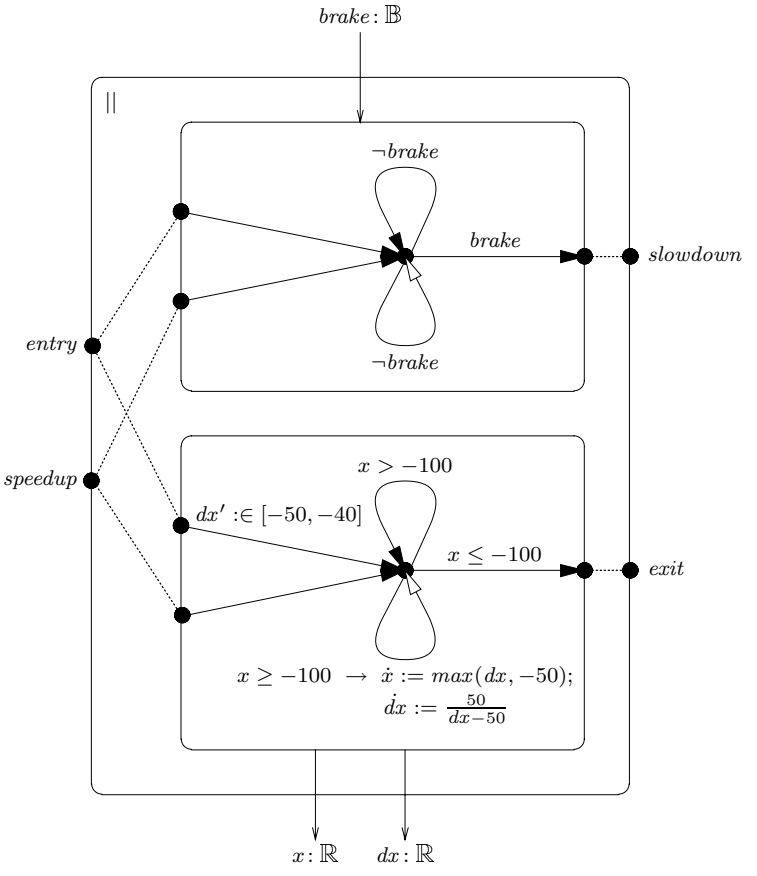

Fig. 9. The component *Drive*

$L^{intf}_{A(J)} = \{from, to\}$.

– The entry condition of *from* is the projection of the jump predicate to the source variables and the primed versions of the uncontrolled sink variables; that is, $\varphi^{en}_{A(J)}(from) = (\exists Z'_J)\, \varphi^{jump}_J$. The entry condition of *to* is unsatisfiable; that is, $\varphi^{en}_{A(J)}(to) = false$.

The *executions of the component* $A(J)$ are defined as follows: the pair $(a, \boldsymbol{w})$ is an execution of $A(J)$ iff $a = from$ and the trace $\boldsymbol{w}$ consists of a single jump (p, q) such that the jump predicate φ^{jump}_J is true if each source variable $x \in X_J$ is assigned the value $p(x)$, and each primed sink variable $y' \in Y'_J \cup Z'_J$ is assigned the value $q(y)$. Moreover, the triple $(a, \boldsymbol{w}, b)$ is an execution of $A(J)$ iff the pair $(a, \boldsymbol{w})$ is an execution of $A(J)$, and $b = to$. In other words, the traces of the atomic discrete component $A(J)$ are the jumps that satisfy the jump predicate φ^{jump}_J. From any given source, there may be no such jumps or there may be several.

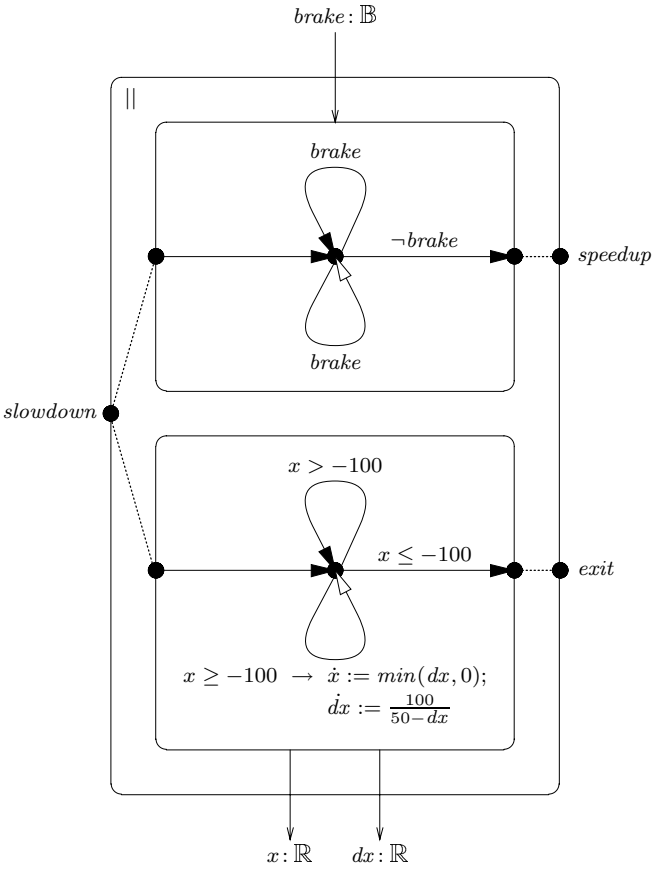

Fig. 10. The component *Halt*

Atomic continuous components The *hybrid components* are built from both atomic discrete components and atomic continuous components using the six operations on components. Each *atomic continuous component* is specified by a flow action. A *flow action* F consists of a finite set X_F of *source variables*, a finite set Y_F of *uncontrolled flow variables* of type $\mathbb{R}$, a finite set Z_F of *controlled flow variables* of type $\mathbb{R}$ disjoint from Z_F, and a predicate φ_F^{flow} on the variables in $X_F \cup \dot{Y}_F \cup \dot{Z}_F$, where $\dot{V}$ is the set of dotted versions of the variables in V. We use the notation $\dot{V}$ only if all variables in V have type $\mathbb{R}$, with the intent that the dotted variable $\dot{x} \in \dot{V}$ represents the first derivative of $x \in V$. The predicate φ_F^{flow} is called *flow predicate*; it is typically written as a guarded differential equation. The flow action F specifies the component $A(F)$. The *interface of the component* $A(F)$ is defined as follows:

– The inputs to $A(F)$ are the source variables of F which are not controlled flow variables, together with the uncontrolled flow variables; that is, $V_{A(F)}^{in} =$

$(X_F \backslash Z_F) \cup Y_F$.
- The outputs of $A(F)$ are the controlled flow variables of F; that is, $V^{out}_{A(F)} = Z_F$.
- Each controlled flow variable depends on each uncontrolled flow variable; that is, for all $x \in V^{in,out}_{A(F)}$ and $y \in V^{out}_{A(F)}$, define $x \prec_{A(F)} y$ iff $x \in Y_F$ and $y \in Z_F$.
- The component $A(F)$ has two interface locations, say, *from* and *to*; that is, $L^{intf}_{A(F)} = \{from, to\}$.
- The entry conditions of *from* and *to* are unsatisfiable; that is, $\varphi^{en}_{A(F)}(from) = \varphi^{en}_{A(F)}(to) = false$. This ensures that jumps take precedence over flows, in the sense that if a component A wishes to jump and concurrently another component B wishes to flow, then the parallel composition $A||B$ will jump.

The *executions of the component* $A(F)$ are defined as follows: the pair $(a, \boldsymbol{w})$ is an execution of $A(F)$ iff $a = from$ and the trace w consists of a single flow (δ, f) such that for all reals $\varepsilon \in [0, \delta]$, the flow predicate φ^{flow}_F is true if each source variable $x \in X_F$ is assigned the value $f(\varepsilon)(x)$, and each dotted flow variable $\dot{y} \in \dot{Y}_F \cup \dot{Z}_F$ is assigned the value $f'(\varepsilon)(y)$, where f' is the first derivative of f. Moreover, the triple $(a, \boldsymbol{w}, b)$ is an execution of $A(F)$ iff the pair $(a, \boldsymbol{w})$ is an execution of $A(F)$, and $b = to$. In other words, the traces of the atomic continuous component $A(F)$ are the flows that at all times satisfy the flow predicate φ^{flow}_F. From any given source and duration, there may be no such flows or there may be several. If there is a flow of a given duration, then there is a flow for each shorter duration as well.

Example The Figures 1–10 illustrate parts of a component which models the control of a railway crossing. In the figures we use the following conventions. Components are represented by rectangles. Input and output variables are represented, repectively, by arrows to and from component boundaries. Locations are represented by little black disks, and between locations, jump actions are represented by arrows with solid (black) points, and flow actions are represented by arrows with hollow (white) points. Interface locations are drawn on component boundaries. Variables which are identified by renaming are connected by solid lines; locations which are identified by renaming are connected by dotted lines. The event type $\mathbb{E}$ is similar to the boolean type $\mathbb{B}$, except that if a variable x has type $\mathbb{E}$, then it is of interest when the value of x changes (from *true* to *false*, or vice versa) whereas the actual value of x at any time is irrelevant. If x has type $\mathbb{E}$, then we write $x!$ for $x' := \neg x$ (to issue an event x), and $x?$ for $x' \neq x$ (to query the presence of an event x). Instead of using jump and flow predicates, we annotate jump and flow actions with guarded commands, because they allow us to omit specifying that a variable is left unchanged. Specifically, by default, an omitted guard is *true*, an omitted list of assignments is empty, the default jump assignment is $x' := x$, and the default flow assignment is $\dot{x} := 0$.

The component *RailCrossing* has three real outputs, the distance x of the train from the crossing, and the positions y_1 and y_2 of the two gates. The boolean input *obstacle* indicates whether or not the driver of the train sees an obstacle

on the crossing, in which case she will try to stop the train. The component *RailCrossing* is the parallel composition of three subcomponents, the train *Train*, the gate mechanics *Gate*, and the gate controller *Control*. We will look only into the component *Train*, which communicates with the gate controller via the output events *approach* and *leave*, and the input events *stop* and *go* (for example, if the gate fails, the gate controller may signal the train to stop). The component *Train* is the serial composition of four subcomponents: the component *Far* controls the speed $\dot{x}$ of the train when it is more than 1000 meters from the gate; an unnamed component issues the event *approach* when the train is at 1000 meters from the gate; the component *Near* controls the speed $\dot{x}$ of the train when it is between 1000 and -100 meters from the gate; and an unnamed component issues the event *leave* when the train is at -100 meters from the gate. The component *Far* holds the speed of the train between 40 and 50 meters per second. The component *Near* is the parallel composition of three subcomponents, *Radio*, *Brake*, and *Engine*. The component *Radio* translates *stop* and *go* events received from the gate controller into a boolean output *remote*, which causes the train to brake. The component *Brake* is an OR gate which computes the boolean disjunction *brake* of the two brake signals *remote* and *local*, where the latter is issued by the driver when she sees an obstacle. The component *Engine* controls the acceleration $\dot{dx} = \ddot{x}$ of the train. It does so by switching between the component *Drive*, which accelerates the train to 50 meters per second, and the component *Halt*, which causes the train to stop. The switching between *Drive* and *Halt* is controlled by the boolean input *brake*, and occurs through the locations *slowdown* and *speedup*. No matter whether the train is accelerating or braking, as soon as it is 100 meters past the gate, the component *Near* relinquishes control.

Acknowledgments Concurrent and sequential hierarchies have long been nested in informal and semiformal ways (e.g., Statecharts [Har87], UML [BRJ99]). While these languages enjoy considerable acceptance as good engineering practice, they do not support compositional formal analysis. The author was pointed to the importance of heterogeneous hierarchies, and the lack of adequate formalization, by Edward Lee and the Ptolemy project at UC Berkeley [DGH+99]. The proposed solution, Masaccio, is built by combining what the author believes are the key ingredients for achieving concurrent, sequential, and timed compositionality: much of the way Masaccio handles parallel composition is borrowed from Reactive Modules [AH99], which in turn build on other synchronous languages such as Esterel [BG88] and Signal [BlGJ91]; many ideas for structuring serial composition are inspired by the work of Rajeev Alur and Radu Grosu [AG00]; the formalization of hybrid executions using the dichotomy between jumps and flows is due to the research around Hybrid Automata [MMP92, ACH+95, AH97b]. Mixed discrete-continuous dynamics have been previously combined with both synchronous [AH97a] and semisynchronous [LSVW96] concurrency models; these settings, however, do not support the nesting of concurrency and sequencing. Related hybrid languages, which focus on simulation rather than mathematical semantics, include Shift [DGV96] and Charon

[AGH+00]. The author is grateful to the Fresco (Formal REal-time Software COmponents[4]) group at UC Berkeley, namely, Luca de Alfaro, Ben Horowitz, Ranjit Jhala, Rupak Majumdar, Freddy Mang, Christoph Meyer, Marius Minea, and Vinayak Prabhu, for many challenging and productive discussions.

References

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[AG00] R. Alur and R. Grosu. Modular refinement of hierarchic reactive machines. In *Proceedings of the 27th Annual Symposium on Principles of Programming Languages*, pages 390–402. ACM Press, 2000.

[AGH+00] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular specification of hybrid systems in Charon. In *HSCC 00: Hybrid Systems—Computation and Control*, Lecture Notes in Computer Science 1790. Springer-Verlag, 2000.

[AH97a] R. Alur and T.A. Henzinger. Modularity for timed and hybrid systems. In *CONCUR 97: Concurrency Theory*, Lecture Notes in Computer Science 1243, pages 74–88. Springer-Verlag, 1997.

[AH97b] R. Alur and T.A. Henzinger. Real-time system = discrete system + clock variables. *Software Tools for Technology Transfer*, 1:86–109, 1997.

[AH99] R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.

[BG88] G. Berry and G. Gonthier. The synchronous programming language Esterel: design, semantics, implementation. Technical Report 842, INRIA, 1988.

[BlGJ91] A. Benveniste, P. le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the Signal language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[DGH+99] J. Davis, M. Goel, C. Hylands, B. Kienhuis, E.A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, and Y. Xiong. Overview of the Ptolemy project. Technical Report UCB/ERL M99/37, University of California, Berkeley, 1999.

[DGV96] A. Deshpande, A. Gollu, and P. Varaiya. Shift: a formalism and a programming language for dynamic networks of hybrid automata. In *Hybrid Systems V*, Lecture Notes in Computer Science 1567. Springer-Verlag, 1996.

[Har87] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[LSVW96] N.A. Lynch, R. Segala, F. Vaandrager, and H.B. Weinberg. Hybrid I/O Automata. In *Hybrid Systems III*, Lecture Notes in Computer Science 1066, pages 496–510. Springer-Verlag, 1996.

[MMP92] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In *Real Time: Theory in Practice*, Lecture Notes in Computer Science 600, pages 447–484. Springer-Verlag, 1992.

[4] For the latest on Fresco activities, see `www.eecs.berkeley.edu/~fresco`.