Formally Ensuring Time Constraints in a Development Process

Ilias Garnier, Christophe Aussaguès, Vincent David
CEA, LIST, Embedded Real Time Systems Laboratory
Point Courrier 94, Gif-sur-Yvette, F-91191 France
Firstname.Lastname@cea.fr

Guy Vidal-Naquet
SUPELEC Systems Sciences (E3S)
Computer Science Department
91192 Gif-sur-Yvette Cedex, France
Guy.Vidal-Naquet@supelec.fr

Abstract

This paper presents a method for the development of systems composed of communicating components that satisfy time constraints, by stepwise refinement. The main result is the formal proof of correctness of a refinement with time constraints. The method is illustrated by some examples. It is issued from studies on real-time embedded systems, but should apply to the specification of other execution-related metrics and is compatible with other development methods which focus on different criteria.

1 Introduction

One particularly important aspect of the development process of systems is formally coupling design and verification of timing constraints. Such constraints, that concern the whole system, must be broken into smaller constraints for each subsystem – a process which when done by hand is difficult and error-prone. Refinement-based development methods are particularly well-suited to this task. In this paper we propose such a formalism, building on Moore machines (Moore 1956), that allows to build systems under timing constraints bearing on both communications and subsystems. If the specification used in a development process is compatible with our formalism, i.e. a specification can be automatically translated into such a machine, then correctness of refinement w.r.t. constraints can be proved, and modules can be developed independently.

Related work. There are many formal methods for specifying and developing systems made of communicating agents. They address a wide range of problems: specifying the concurrent aspects (Bolognesi and Brinksma 1987; Sifakis 2009), the algorithmic aspects (Nielsen et al. 1988) and the temporal aspects (Alur and Dill 1994) of the systems; helping the developer by enforcing incremental methods, and verifying the correctness of the result with regard to the specification with a varying degree of automation.

Our work was inspired by the specification and verification of real-time systems (David et al. 1998). A compositional verification of systems using Moore machines was given by (Clarke, Long, and McMillan 1989). Our specification formalism based on a notion of reaction time was inspired by the literature on information flow analysis (Barbuti, Bernardeschi, and De Francesco 2002) and by the literature on the category-theoretic view of process algebras (Meng and Barbosa 2006).

2 Enforcing constraints in refinement-based development

Our specification formalism models programs as finite-state transducers known as Moore machines (Moore 1956). Timed Moore machines are then defined as Moore machines enriched with information on reaction time. They will be abbreviated TM-machines or simply machines when the context is unambiguous. We then give their composition laws and the definition and properties of their refinement.

2.1 Timed Moore machines

In the rest of the paper, we will assume the existence of some basic data types, among which finite integers **int** and booleans **bool** = $\{$**tt** ; **ff** $\}$. The set of data types is the free monoid $\langle Type, \times\rangle$ generated by these finite sets and closed under cartesian product. Thus, our data will be vectors of fixed dimension. The neutral element of $Type$ is the singleton set $\mathbb{1} = \{\star\}$ (up to isomorphism). We will work up to the isomorphisms $(a,b) \mapsto (b,a)$, $(\star,x) \mapsto x$, $(x,\star) \mapsto x$ and associativity. These isomorphisms can be extended to sequences of data (words), set of sequences of data (languages) and machines. Moreover, the pairing operation $\langle a,b\rangle$ can be extended to words of same length. As a model of discrete duration, we will use the order on integers.

Moore Machines. We will first proceed to the definition of a plain Moore machine. A Moore machine $m = \langle IN, OUT, Q, \Gamma, E\rangle$ is defined as follows.

- The input and output alphabets are $IN, OUT \in Type$.
- The finite set of states Q (with a distinguished initial state q_{init}) is labelled by $\Gamma : Q \to OUT$ which associates to each state its output data vector.
- The finite set of edges is $E \subseteq Q \times IN \times Q$. We note $p \overset{i}{\to} q$ as a shorthand for $(p,i,q) \in E$. We also require that all possible inputs must be accepted (this will be refered to as the *totality* constraint).

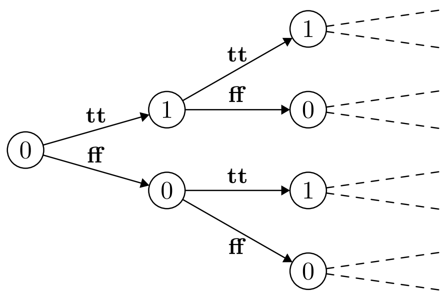
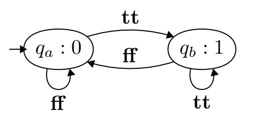


Figure 1: Example of unfolding.

This defines a Moore machine whose inputs and outputs are tuples of fixed dimension. The positions on the tuples are called informally "ports". The set of Moore machines on alphabets IN and OUT is noted $IN \Rightarrow OUT$. For instance, a machine that has a couple of integers on input and boolean for output has IN = **int** × **int**; OUT = **bool**. Input data will be noted $\langle i_0, i_1 \rangle \in IN$, and output symbols $b \in OUT$.

The following figure is a first example of machine such that IN = **bool** and $OUT = \{0, 1\}$. The output data is drawn inside nodes alongside the state name $q_{a,b}$, and the input is displayed on edges.



This machines outputs the symbol 1 when it receives **tt** , and it outputs 0 when it receives **ff** .

Run of a machine. A run is defined as in the case of a plain non-deterministic finite-state machine. We recall here some definitions. Let E be a set of edges and Q its underlying set of states. A finite run is a word of the shape $q_0 in_0 q_1 in_1 \ldots in_n q_n$ belonging (roughly) to the transitive closure E^+ of the transition relation E. Such a run defines an output word $\Gamma(q_0).\Gamma(q_1) \ldots \Gamma(q_n)$ associated to the input word $in_0 \ldots in_n$. We will note $q_0 \xrightarrow[w_o]{w_i}{}^+ q_n$ a run with input word w_i and output word w_o going from q_0 to q_n. Let $Runs_E : Q \times IN^* \to \wp_{fin}(E^+)$ the function which associates to any state the set of maximal runs associated to a finite sequence of inputs (for a fixed set of edges E).

Unfoldings. As a mean to display our definitions in a more graphic way, we will use unfoldings (also called synchronization trees (Winskel 1984)) as canonical representatives of bisimulation classes. Informally, the infinite unfolding of a machine from a state q is a finitely branching, infinitely deep tree whose nodes correspond to states and whose edges are the possible transitions between states. Thus, all finite branches of an unfolding are finite runs and given an input word, $Runs_E$ computes a sub-tree of the unfolding. Fig. 1 shows the unfolding of the machine depicted before.

Behavior of a machine In order to specify and verify systems, it is essential to have a formal model which abstracts away irrelevant details. Two different systems which perform the same computation under the same constraints should be equivalent in this model. Two such systems have the same *behavior*. The classical way of defining equivalence of state-transition machines is bisimulation.

Definition 1 (Bisimulation) *Let $m : I \Rightarrow O$ be a machine. Let p and q be two states of m. p and q are said to be bisimilar if and only if $p \sim q$. $\sim$ is the greatest relation s.t.:*

$$\begin{aligned} p \sim q \quad \leftrightarrow \quad & \Gamma(p) = \Gamma(q) \wedge \\ & (\forall p \xrightarrow{a} p', \exists q \xrightarrow{a} q', p' \sim q') \wedge \\ & (\forall q \xrightarrow{a} q', \exists p \xrightarrow{a} p', p' \sim q') \end{aligned}$$

The *behavior* of a machine in a given state is defined as the bisimulation equivalence class it belongs to. Although not technically useful for bisimulation, the ability to quantify differently over the data coming from different ports will be crucially used when treating the negation of bisimulation.

Reaction time. Specification of reaction time guarantees a *functionally dependent* reaction to an input in bounded time. Many verification methods only check for the occurrence of some particular events (e.g. sending a message) after receiving an input – property which is not sufficient to entail a functional dependency on a critical data path. As a trivial example, consider a program which periodically emits a randomly generated message; or another program which ignores some of its incoming orders. We make functional dependency formally explicit and study how to bound the time of a functionally dependent output. In this paper, this idea is applied to Moore machines, but the concept of reaction time could be investigated in other formalisms, such as timed automata (Alur and Dill 1994; David et al. 2010).

Reaction to an input is a purely observational quality of a system, and should not rely on particular implementation methods. We will first define reaction to an input; this definition will then be enriched to take into account a bound on the number of transitions needed to observe a reaction. A extensive study of these concepts can be found in (Garnier et al. 2011).

Discriminating states and inputs. Characterizing constantness for a total function f is straightforward: if the result of applying its whole input domain results in a single point in the codomain, we know that f is constant. Dually, if there exists two inputs $i_1 \neq i_2$ such that $f(i_1) \neq f(i_2)$ then f is non-constant. We thus define *discriminating reaction to inputs* as *non-constantness w.r.t. an input*, i.e. *functional dependency*. However, in the setting of automata theory this corresponds to the negation of bisimilarity.

Definition 2 (Non-bisimilarity) *Let $m : I \Rightarrow O$ be a machine. Let p and q be two states of m. p and q are said to be non-bisimilar if and only if $p \nsim q$, where $\nsim$ is the following inductively defined relation:*

$$\text{BASE}\ \frac{\Gamma(p) \neq \Gamma(q)}{p \nsim q}$$

$$\text{IND}\ \frac{\begin{array}{c}\exists p \xrightarrow{a} p', \forall q \xrightarrow{a} q', p' \nsim q' \ \vee\ \exists q \xrightarrow{a} q', \forall p \xrightarrow{a} p', p' \nsim q' \\ \Gamma(p) = \Gamma(q)\end{array}}{p \nsim q}$$

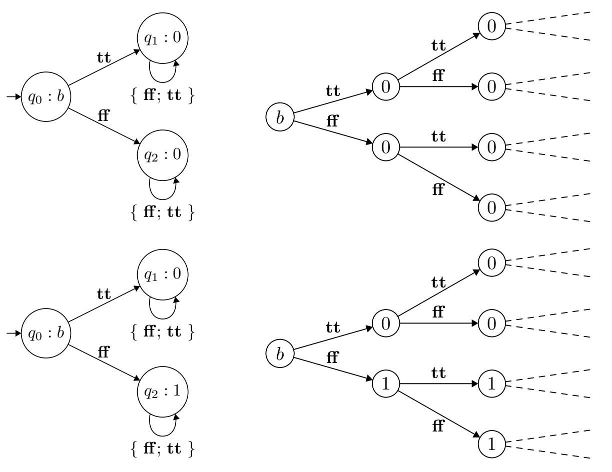


Figure 2: Non-discriminating and discriminating states.

The extension to multiple ports can be found in Sec. 2.1.

A machine can be seen as a relation extended in time (Abramsky 1996). We can thus write a very similar definition for the predicate RQ which characterizes states which are discriminating w.r.t. their next input: if there exists two different inputs such that the reached states are not equivalent, the state is discriminating.

Definition 3 (Discriminating states, separating pairs) *Let $m : I \Rightarrow O$ be a machine. Let r be a state of m. The fact that r is reactive is noted $RQ(r)$.*

$$RQ(r) \triangleq \left\{ r \mid \exists in_p \neq in_q \in I, (\exists r \overset{in_p}{\to} p, \forall r \overset{in_q}{\to} q, p \nsim q) \right\}.$$

Such input symbols (in_p, in_q) are the separating pairs *of state r. A separating pair of a state r is deterministic iff:*

$$\exists in_p \neq in_q \in I, (\forall r \xrightarrow{in_p} p, \forall r \xrightarrow{in_q} q, p \nsim q).$$

It will be shown that deterministic separating pairs are preserved by the refinement operation. The set of separating pairs of a state r is noted $\mathcal{SP}(r)$, and the set of deterministic separating pairs is $\mathcal{DSP}(r)$. Given two states p, q, $(in_1, in_2) \in \mathcal{SP}(r) \cap \mathcal{SP}(r)$ is a strongly separating pair *of (p, q) (noted $\mathcal{SSP}(p, q)$) iff:*

$$(\exists p \overset{in_1}{\to} p', \forall q \overset{in_2}{\to} q', p' \nsim q') \vee (\exists p \overset{in_2}{\to} p', \forall q \overset{in_1}{\to} q', p' \nsim q').$$

Having $(in_1, in_2) \in \mathcal{SSP}(p, q)$ implies that (in_1, in_2) is also a separating pair of the set-theoretical union of the transitions of p and q. The subset constituted of deterministic separating pairs is noted $\mathcal{DSSP}(p, q)$.

Fig. 2 shows two machines. In the topmost one, the initial state q_0 has all its successor states in the same bisimulation class. Hence, any input data read from q_0 will have no impact on the future of the execution of this machine. In the lowest one, this is not the case; state q_0 discriminates its inputs, thanks to the strongly separating pair (**tt**, **ff**). In the rest of the paper, as an abuse of language, we will use the term "reactive" instead of "discriminating" since it conveys the core of our idea.

Reaction time and separators. Reaction time is a finer-grained notion of reactiveness. Let $m : I \Rightarrow O$ be a machine. Let's assume that the state r of m is reactive. This implies the existence of at least two transitions $r \xrightarrow{in_p} p$ and $r \xrightarrow{in_q} q$ s.t. $in_p \neq in_q$ and $p \nsim q$. Any constructive proof of $p \nsim q$ yields a pair of separating runs run_p, run_q labelled by an input word w s.t. $run_p \in Runs_E(r, in_p.w)$ and $run_q \in Runs_E(r, in_q.w)$:

$$run_p = r \xrightarrow[out_p.o_p]{in_p.w}{}^{+} p_t \quad run_q = r \xrightarrow[out_q.o_q]{in_q.w}{}^{+} q_t.$$

These runs are such that $o_p \neq o_q$. The input word w is called a *separator*, and we say that p and q are *separable*. The pair (o_p, o_q) is an *observable effect* induced by the corresponding separating run at time $|w|$. This notion has a graphic representation in terms of unfoldings: it amounts to performing a bounded-depth comparison on sub-trees. Fig. 3 shows two unfoldings of two different states. The fact that they are non-bisimilar is proved by the existence of two separators (although one would suffice) of length two and three, as emphasized by the dotted paths. The observable effects are respectively $(8, 3)$ at time 2 and $(1, 3)$ at time 3.

A separator w is *deterministic* when all its runs $run_p \in Runs_E(r, in_p.w)$ and $run_q \in Runs_E(r, in_q.w)$ are separating (i.e. correspond to a proof of $p \nsim q$). The set of deterministic separators of p and q will be noted $\mathcal{DS}(p, q)$ A pair of states (p, q) is *strongly separable* if all infinite input words are prefixed by a deterministic separator of (p, q). This implies the occurrence of an observable effect in finite time. Together with the fact that the set of edges is finite, this implies the finiteness of $\mathcal{DS}(p, q)$. The fact that (p, q) are strongly separable will be noted $p \;)\!(\; q$.

Let p, q be two states s.t. $p \;)\!(\; q$. In this setting, an observable effect (o_p, o_q) at time t is *deterministic* iff all input words of w of length t generate this observable effects, i.e. all runs are s.t.:

$$p \xrightarrow[out_p.o_p]{w}{}^{+} p_t \quad q \xrightarrow[out_q.o_q]{w}{}^{+} q_t.$$

This defines a partial function $diff_O : Q \times \mathbb{N} \rightharpoonup O \times O$ which possibly maps an deterministic observable effect to a state and a number of transitions. In a similar fashion, it is possible to define a function $DSPseq_I : Q \times \mathbb{N} \to \wp_{fin}(I \times I)$ associating a set of deterministic strongly separating pairs to a state and a number of transitions. $DSPseq_I(q, t)$ is defined by computing the intersection of all the deterministic strongly separating pairs reachable in t transitions from q (Garnier et al. 2011). Together, $diff_O$ and $DSPseq_I$ compute a deterministic and linear under-approximation of the behavior of a machine.

We can now define the notion of reaction time of a state.

Definition 4 (Reaction time) *Let $m : I \Rightarrow O$ be a machine. The reaction time of a state w.r.t. an input is the* maximum *number of transitions that must be performed to see the first observable effect arise. Let $q \in Q$ be a state s.t. $RQ(q)$ holds and s.t. for any separating pair of inputs to a pair of states (q_1, q_2), we have $q_1 \;)\!(\; q_2$. We note $RT(q) = t$ the fact*

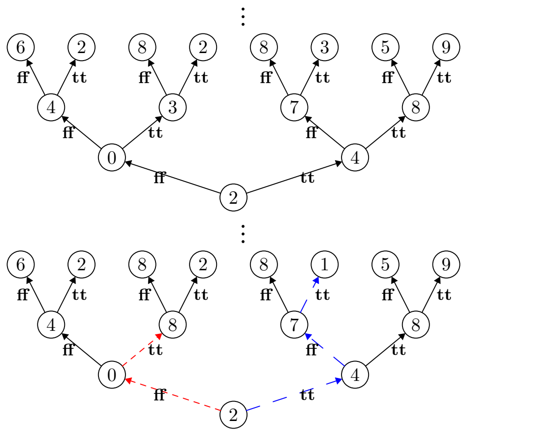


Figure 3: Example of bounded non-bisimilarity on unfoldings.

that q has a reaction time of t transitions, where:

$$RT(q) = max\{|w| \mid (a_1, a_2) \in \mathcal{DSP}(q),\ q \overset{a_1}{\to} q_1,\ q \overset{a_2}{\to} q_2,\ w \in \mathcal{DS}(q_1, q_2)\}$$

Multiple input-output ports. The previous definitions only address the case of machines whose interface is structured in one input port and one output port. We can generalize these definitions to a more general setting. Let $m : I_1 \times I_2 \Rightarrow O_1 \times O_2$ be a machine with input and output ports structured such that we only want to consider the reaction time (i.e. the deterministic separators) from I_1 to O_1. A deterministic separator $w \in \mathcal{DS}^{I_1,I_2}$ can either be *existential*, if there exists a word on I_2 s.t. w exists; or *universal*, if w exists for all words on I_2. Both kinds are useful, but in order to ensure compositionality we are interested in universal deterministic separators.

However, enumerating all contexts in order to find these separators is cumbersome. Working on the *projection* of the relevant data on the whole machine (i.e. when erasing the data on irrelevant ports) would be more lightweight. Projection on machines is defined as the extension of projections on data. If $m = \langle A \times I, B \times O, Q, \Gamma, E\rangle$, then the projection of m on $I \Rightarrow O$ is defined as

$$\pi_{I \Rightarrow O}(m) = \langle I, O, Q, \pi_O \circ \Gamma, \{(p, \pi_I(i), q) | (p, i, q) \in E\}\rangle.$$

Deterministic separators on the projection are a *subset* of the universal deterministic separators. We can thus *soundly* restrict our attention to machines with one input port and one output port. The proof is out of the scope of this paper.

2.2 Composition of machines

Given two machines with their respective types, there are at least two ways to present their composition: we can either give explicitly a communication network as a graph, or we can give an algebra of composition operations on machines. We choose the latter solution. (Clarke, Long, and McMillan 1989) use explicit port names and their composition operation performs both parallel composition and communication. We opt for a more category-theoretical view (Abramsky 1996). Our operations are parallel composition and feedback (sequential composition can be recovered from feedback). These operations allows us to build more complex systems from simpler ones. In (Garnier et al. 2011), we have shown that reaction time is not compositional in general, but that a sound under-approximation of it can be computed and easily composed. This under-approximation relies on deterministic separators and deterministic separating pairs (which are themselves under-approximated by deterministic separators on projections). We will thus consider these notions in the following developments. We will only sketch the process of composition, as the formal definitions and associated proofs can be found in (Garnier et al. 2011).

Parallel composition. Let $m_1 : A_1 \Rightarrow B_1$ and $m_2 : A_2 \Rightarrow B_2$ be two machines:

$$m_{x \in \{1,2\}} = \langle A_x, B_x, Q_x, q_{init,x}, \Gamma_x, E_x\rangle.$$

The parallel composition is $m_1 \parallel m_2 : A_1 \times A_2 \Rightarrow B_1 \times B_2$. It is computed as a standard synchronous product of automata: there is no inter-communication between the two machines. The resulting signatures are the cartesian product of the sub-components. Parallel composition does not alter the temporal behavior of its sub-components. The machine $m_1 \parallel m_2$ is computed as follows.

$$\begin{aligned} &Q_\parallel = Q_1 \times Q_2, \quad q_{init,\parallel} = (q_{init,1}, q_{init,2}) \\ &E_\parallel = \{((q_{11}, q_{21}),\ \langle input_1, input_2\rangle,\ (q_{21}, q_{22})) \mid \\ &\qquad (q_{1i}, input_i, q_{2i}) \in E_i\} \\ &\Gamma_\parallel(q_1, q_2) = \langle \Gamma_1(q_1), \Gamma_2(q_2)\rangle \end{aligned}$$

Feedback. The feedback operation redirects the data of an output port to an input port. By creating new computation paths in the composite machine, the feedback operation can alter its reaction time. For instance, if there is no separator on an input-output port pair (e.g. in the case of a parallel composition), the feedback will allow to compute a new reaction time by summing the reaction time of the start state with the reaction times of the intermediate states (only if their composition guarantees a functional dependency).

Feedback definition. Let $m : I \times U \Rightarrow O \times U$ be a machine:

$$m = \langle I \times U, O \times U, Q_m, q_{init,m}, \Gamma_m, E_m\rangle.$$

The operation of connecting an output of m to an input of identical type U is $feedback_U : (I \times U \Rightarrow O \times U) \to I \Rightarrow O$. Performing the feedback restricts the set of edges to those compatible with the data emitted on the output component U. The feedback is defined below.

$$\begin{aligned} &Q_{fb} = Q_m \quad q_{init,fb} = q_{init,m} \quad \Gamma_{fb} = \Gamma_m \\ &E_{fb} = \{(q, \pi_A(input), q') \in E_m \mid \pi_U(input) = \pi_U(\Gamma_m(q))\} \end{aligned}$$

Observe that the totality constraint (cf. Sec. 2.1) is still respected on the resulting machine: the feedback operation only reduces non-determinism on $I \Rightarrow O \times U$. Hence, previous *deterministic* separators on $I \Rightarrow O$ still exist, as well as deterministic separating pairs. New separators on $I \Rightarrow O$ can be computed by composing separators from $I \Rightarrow U$ with separators from $U \Rightarrow O$. This process of composition relies on matching deterministic observable effects with deterministic strongly separating pairs.

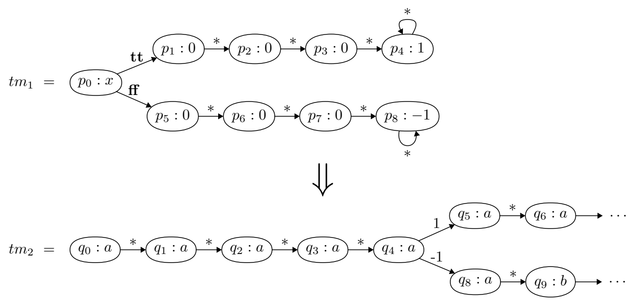


Figure 4: Reaction time computation.

Definition 5 (Composing observable effects) *Let q be a state of $feedback_U(m)$. Let t be a number of transitions s.t. $diff_U(q,t) = (o_1, o_2)$ and s.t. $(o_1, o_2) \in DSPseq_I(q, t+1)$. Then, for all q' reachable in t transitions (i.e. $q \rightarrow^{t+1} q'$), the observable effects common to all these states q' also belong to q.*

Example of feedback. Using feedback and parallel composition, it is possible to express sequential composition, as shown in the following example. Let m_1 and m_2 be two machines (cf. Fig. 4). The state p_0 of m_1 is reactive with (**tt** , **ff**) as deterministic separating pair and $diff_{int}(p_0, 3) = (1, -1)$. Thus, the reaction time of p_0 is 3. The set of (deterministic) separators for p_1 and p_5 is all input words of length 3, that is $bool^3$. Thus, we have $p_1 \mathrel{)(} p_5$.

In m_2, the states q_0 to q_3 of m_2 are not reactive. State q_4 is reactive and has a reaction time of 1 for the deterministic separating pair $(1, -1)$, with as separators all words of length 1 with input alphabet int. Thus, $q_5 \mathrel{)(} q_8$ and we have as only observable effect $diff_{\{a,b\}}(q_4, 1) = (a, b)$. The sequence of deterministic strongly separating pairs of p_0 is

$$DSPseq_{int}(q) = \varnothing.\varnothing.\varnothing.\varnothing.\{(1,-1)\}.\varnothing\ldots$$

We have trivially $diff_{int}(p_0, 3) \in DSPseq_{int}(q, 4)$, hence we obtain $diff_{\{a,b\}}(p_0, 5) = (a, b)$ as compound observable effect. We obtain the words of length $|bool^3| + |bool^2| = 5$ as fresh deterministic separators.

Definition of the refinement relation. Our refinement relation is a variation of *simulation*. We want a relation such that whenever a machine m_1 is refined by a machine m_2, reactivity is conserved. Unfortunately, plain separators are *not* conserved by refinement, as well as separating pairs. The reason is that refinement can break asymmetries arbitrarily by suppressing transitions, making previously non-bisimilar states bisimilar.

However, the totality constraint guarantees the conservation of the deterministic counterparts of the aforementioned notions (cf. the universal quantifications in the definitions). Since our definition of reaction time $RT(q)$ is defined in terms of deterministic separators and deterministic separating pairs, we can show that it is indeed preserved in a strong sense by refinement: it monotonically decreases.

Let $m_{1,2} = \langle IN, OUT, Q_{1,2}, q_{init,1,2}, \Gamma_{1,2}, E_{1,2}\rangle$ be two machines of identical types. The formal definition of the refinement relation $\precsim: Q_1 \times Q_2$ follows.

Definition 6 (Refinement) *We say that $q_1 \in Q_1$ refines $q_2 \in Q_2$, noted $q_1 \precsim q_2$, when:*

$$\Gamma_a(q_1) = \Gamma_b(q_2) \;\wedge\; \forall q_1 \xrightarrow{input} p_1, \exists q_2 \xrightarrow{input} p_2, p_1 \precsim p_2.$$

Refinement is straightforwardly extended to machines: m_b refines m_a, noted $m_a \precsim m_b$, iff $q_{init,a} \precsim q_{init,b}$. Note that the validity of refinement is decidable. The proof that reaction time can only decrease with refinement follows directly from the fact that deterministic observable effects are preserved (by definition of deterministic observable effects, all input words, hence all runs, generate them).

Refinement is a congruence. When the developer performs a refinement, it is often a local modification of an earlier design. It may be too costly to recompute the whole machine (by performing the parallel compositions and feedbacks) just to check the correctness of this local modification. We want local modifications to have a local impact, and this behavior arises when the refinement relation is a *congruence*, i.e. it commutes with any context. In our case, the contexts $\mathcal{C}$ for a machine m are defined inductively as:

$$\begin{array}{rcl} C[m] & ::= & m \\ & | & \forall U, feedback_U(C[m]) \\ & | & \forall m', m' \parallel C[m] \\ & | & \forall m', C[m] \parallel m' \end{array}$$

Of course, in this definition, all the well-formedness constraints are supposed respected. We can now state our theorem:

Theorem 1 *Refinement is a congruence.*

$$\forall m_a, m_b,\;\; m_a \precsim m_b \rightarrow \forall C, C[m_a] \precsim C[m_b]$$

The proof proceeds straightforwardly by induction on contexts.

2.3 Specification mechanism

We have defined a notion of reaction time and seen how it is preserved under composition. In order to use it during development, we propose a small specification language relying on reaction time. As noted in Sec. 2.1, reaction time on a input-output port pair (I, O) can be conditional to some other value on an input port I'. The language is presented as a propositional logic where atoms are timing constraints of the shape $Guard \rightarrow I \overset{d}{\rightsquigarrow} O$ where I is an input port, O is an output port d is a duration and $Guard$ is a set of equations of the shape $I' = value$. The guard can of course be empty.

The set of formulas ϕ is defined inductively: $\phi ::= Guard \rightarrow I \overset{d}{\rightsquigarrow} O \mid \phi_0 \wedge \phi_1 \mid \phi_0 \vee \phi_1$. A formula of the form $Guard \rightarrow I \overset{d}{\rightsquigarrow} O$ is true on a state q if and only if $\exists t \le d, q \in RT_{I,O}(t)$ under the additional assumptions of $Guard$. The validity of an arbitrary formula follows straightforwardly.

2.4 Abstraction of machines

In practice, it is difficult to develop and verify machines with large state-space, for algorithmic complexity reasons. It is useful to investigate the behavior of our definitions when we *abstract* our state transition system (Cousot and Cousot 1976).

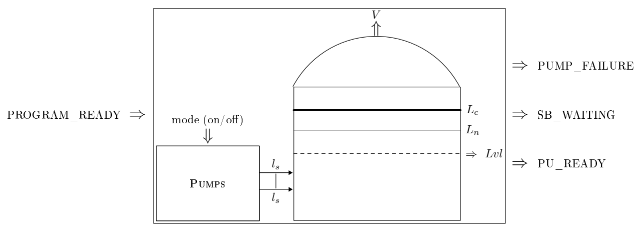


Figure 5: Structure of the physical system.

Abstract alphabets. An abstraction is the data, for each basic datatype Σ_0, of an abstract lattice Σ_1 and of a galois connection $\Sigma_0 \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \Sigma_1$ where $\alpha : \Sigma_0 \to \Sigma_1$ and $\gamma : \Sigma_1 \to \wp_{fin}(\Sigma_0)$. An abstract value denotes a set of concrete values. The order relation of the lattice is the abstraction of set inclusion on the powerset of the concrete alphabets. The various properties of galois connections are explained in e.g. (Schmidt 1995; Cousot and Cousot 1992).

Abstract machines. The abstraction and concretization function on alphabets can be lifted to machines. An abstract machine is a Moore machine using abstract alphabets. Let $\langle IN_1, OUT_1, Q_1, \Gamma_1, E_1\rangle$ be an abstract machine. The corresponding set of concrete machines is:

$$\{\langle IN_0, OUT_0, Q_0, \Gamma_0, E_0 \mid Q_0 = Q_1 \wedge \\ \forall q, \Gamma_0(q) \in \gamma(\Gamma_1(q)) \wedge \\ \forall (p, in_1, q), \forall in_0 \in \gamma(in_1),\ \exists (p, in_0, q) \in E_0\}.$$

Soundness of abstraction w.r.t. reaction time In this setting, an abstract separator w for two abstract states p, q is an abstract word s.t. for all possible abstract machines, the two states are still separable by a word of length less than $|w|$. It can be shown that abstract deterministic separators allow to overapproximate reaction time as defined in our framework. The full formalization is out of scope. We will now assume that we can safely develop and verify using approximated data types (e.g. intervals instead of integers, booleans extend with undefined value, etc).

3 Example: the steam-boiler

This section contains an example of refinement-based development using our formalism. This example comes from the real-time domain, but it is meaningful since it shows a development process for actions that have time constraints. We will first proceed to the definition and explanation of some syntactical extensions, then we will give an informal overview of the problem, and we will proceed to the development itself.

3.1 Syntactical extensions.

We will use explicit port names instead of the positional notation used previously (these are equivalent). While defining machines, We will also use variables which will be assigned on transitions. Assignment of a value v to a variable x will be noted $x := v$. Accessing the value of the variable will be noted $!x$. Also, we will assume that data types are endowed with lattice-like structure (in particular, we will use the semi-lattice of booleans **tt** , **ff** $\leq \top$, meaning that $\top$ is an undefined value). Our definitions are sound relatively to the extensions to lattices. Together, these extensions do not increase the expressive power of our formalism, but they will allow us to reduce the state space of machines.

3.2 Informal specification of the steam-boiler.

Our example is inspired from the specification given in (Abrial 1996). The whole system is composed of a physical system and of a computer program (which we must develop) whose task is to control and monitor the physical system.

Physical components of the steam-boiler. The steam-boiler is a device which inputs water, heats it and outputs steam. It is composed of the parts described thereafter.

- The steam-boiler itself is a water tank characterized by its critical maximal water level L_c, its nominal maximal water level L_n, its current water level Lvl and the amount of steam being produced V. L_c, L_n and Lvl are in litres; V is in litres/unit of time.
  If $Lvl \geq L_c$ for 5 units of time or more, the steam-boiler physical integrity is endangered.
- The water is poured into the steam-boiler by two water pumps whose functioning mode is coupled. Their capacity is each of l_s litres/unit of time. Water pumps can stop functioning, in which case they emit an appropriate message. For simplicity's sake, we assume that the pumps can be turned on or off immediately.

Overview of the control program. The program's main task is to ensure that the level of water does not exceeds L_c. The program must also handle pump failures. It features the following functioning modes:

- initialization (INIT),
- normal functioning mode (NORMAL),
- degraded functioning mode (DEGRADED),
- emergency stop (EMERGENCY STOP).

The INIT mode. The control unit is in an active waiting state until it receives the message SB_WAITING ("steam-boiler waiting") from the physical unit. It then checks that the quantity of steam produced V is equal to 0. If not, it indicates a physical malfunction and the program enters the EMERGENCY STOP mode. If $V = 0$, the control unit sends a message PROGRAM_READY ("program ready") to the physical unit and waits for the message PU_READY ("physical unit ready"). When PU_READY is received, the control units enter the NORMAL functioning mode.

The NORMAL mode. In this mode, the control unit task is to adjust the quantity of water Lvl so that it does not go above the critical level L_c. The target level of water is L_n. The quantity of water is adjusted by turning on or off the pumps. This process is activated periodically. When a message PUMP_FAILURE is received, it means that one of the pumps has ceased functioning, altering the quantity of water flowing into the steam-boiler. In this case, the control unit goes into DEGRADED mode. Whenever the critical L_c level is reached, the control unit goes into EMERGENCY STOP mode.

The DEGRADED mode. This mode is similar to the NORMAL functioning mode, except that it assumes that only one pump is functioning. Whenever another PUMP_FAILURE message is received, or whenever the critical level is reached, the control unit goes into EMERGENCY STOP mode.

The EMERGENCY STOP mode. Entering this mode shuts down the physical system. This is handled by the physical units. This mode is terminal: it is impossible to go to another functioning mode. All these functioning modes are organized as in the following diagram:

$$\begin{tikzcd}
\boxed{\text{Init}} \arrow[r, "\text{init ok}"] \arrow[dr, "\text{init failure}"'] & \boxed{\text{Normal}} \arrow[loop above, "\text{ok}"] \arrow[r, "\text{pump failure}"] \arrow[d, "\text{physical risk}"] & \boxed{\text{Degraded}} \arrow[loop above, "\text{ok}"] \arrow[dl, "\text{physical risk}"] \\
 & \boxed{\text{EmStop}} &
\end{tikzcd}$$

The physical system structure and interface is described in Fig. 5.

3.3 The steam-boiler control program.

We recall the temporal constraints that the system should respect: $i)$ the level of water Lvl must not go above L_c for more that five units of time, $ii)$ the treatment performed by NORMAL and DEGRADED must be periodic, $iii)$ in the rest of the paper, the "immediate" treatment of a message should be understood as having an observable relation as soon as possible. Our example starts with three machines, which are described below.

First development step. Our machines are displayed as automata enclosed in boxes. The input ports are at the left of these boxes, and the output ports are at the right. State names will be omitted when they are not needed. The transitions are labelled by input symbols, i.e. tuples drawn from the alphabet of the machine (similarly for output symbols). Timing constraints are contained in **dashed** nodes. For this first development step, we display explicitly on edges and in states to which port belongs the data. For space reasons, this will not be the case in subsequent refinements.

- INIT handles the initialization. It inputs the boolean messages SB_WAITING and PU_READY , as well as the amount of steam V. It outputs a boolean message PROGRAM_READY to the physical system, a boolean FM_STARTUP message to the FUNCTIONING MODES machine and a boolean message to the EMERGENCY STOP in case of failure during the initialization. As can be seen in Fig. **??**, the starting point of this machine is a state which inputs everything and inputs anything.
- FUNCTIONING MODES handles the NORMAL and DEGRADED modes. It inputs the current quantity of water Lvl, encoded as a floating point value. It also inputs a boolean message FM_STARTUP allowing the machine to start monitoring the physical system as well as a boolean message PUMP_FAILURE indicating a physical failure in a pump. It outputs the functioning $mode$ of the pump as a boolean which is **tt** if the pumps must be started or **ff** if they must be stopped. In case of physical risks, it also emits a boolean message EM_STOP .

  Fig. 6 shows that this initial machine behavior is to wait on the first state for the FM_STARTUP message from INIT. In this waiting state, it emits nothing. When this message is received, the machine proceeds to some still unknown computations, outputting possibly anything.
- EMERGENCY STOP is a software interface to the physically-managed emergency stop. It receives boolean messages EM_STOP and let the physical system handle its shutdown. We assume that all messages emitted to this machine are correctly handled (i.e. no message is missed).

Refinement of INIT. The INIT machine is refined so as to obey to its informal specification. The resulting machine (Fig. 6) first waits for a SB_WAITING message. Upon reception of this message, it immediately checks for the error condition: if $V > 0$, an EM_STOP message is sent. Note that the immediate treatment of this message should be mandatory to the receiver. This condition holds because of our hypothesis on the machine EMERGENCY STOP.

If the error condition is not triggered, the machine emits a PROGRAM_READY message to the physical unit and waits a PU_READY message from it. When this message is received, the INIT machine sends a message to be treated immediately by FUNCTIONING MODES. It then goes into a non-reactive state.

The fact that the original INIT machine is correctly refined by any machine straightforwardly entails that this refinement is correct. Moreover, the fact that mandatory messages are well treated is verified by performing the synchronous product of INIT and FUNCTIONING MODES: all states whose component contains a mandatory message are reactive (observe that the initial state of FUNCTIONING MODES does an active waiting on this very message).

Refinement of FUNCTIONING MODES. We proceed by refining FUNCTIONING MODES into three sub-machines, whose description follows.

- The NORMAL machine handles the normal functioning mode. The machine actively awaits a NORM_STARTUP before proceeding to some non-specified computations.
- The DEGRADED machine handles the degraded functioning mode. The behavior of this machine is almost identical to NORMAL's, it will thus be omitted from our developments.
- The MODESWITCH machine handles the activation of the NORMAL and DEGRADED modes, and also routes the data produced by either one of them to the other machines and the physical units. This machine actively waits for a FM_STARTUP message (sent by INIT). It then sends a NORM_STARTUP message to activate the NORMAL

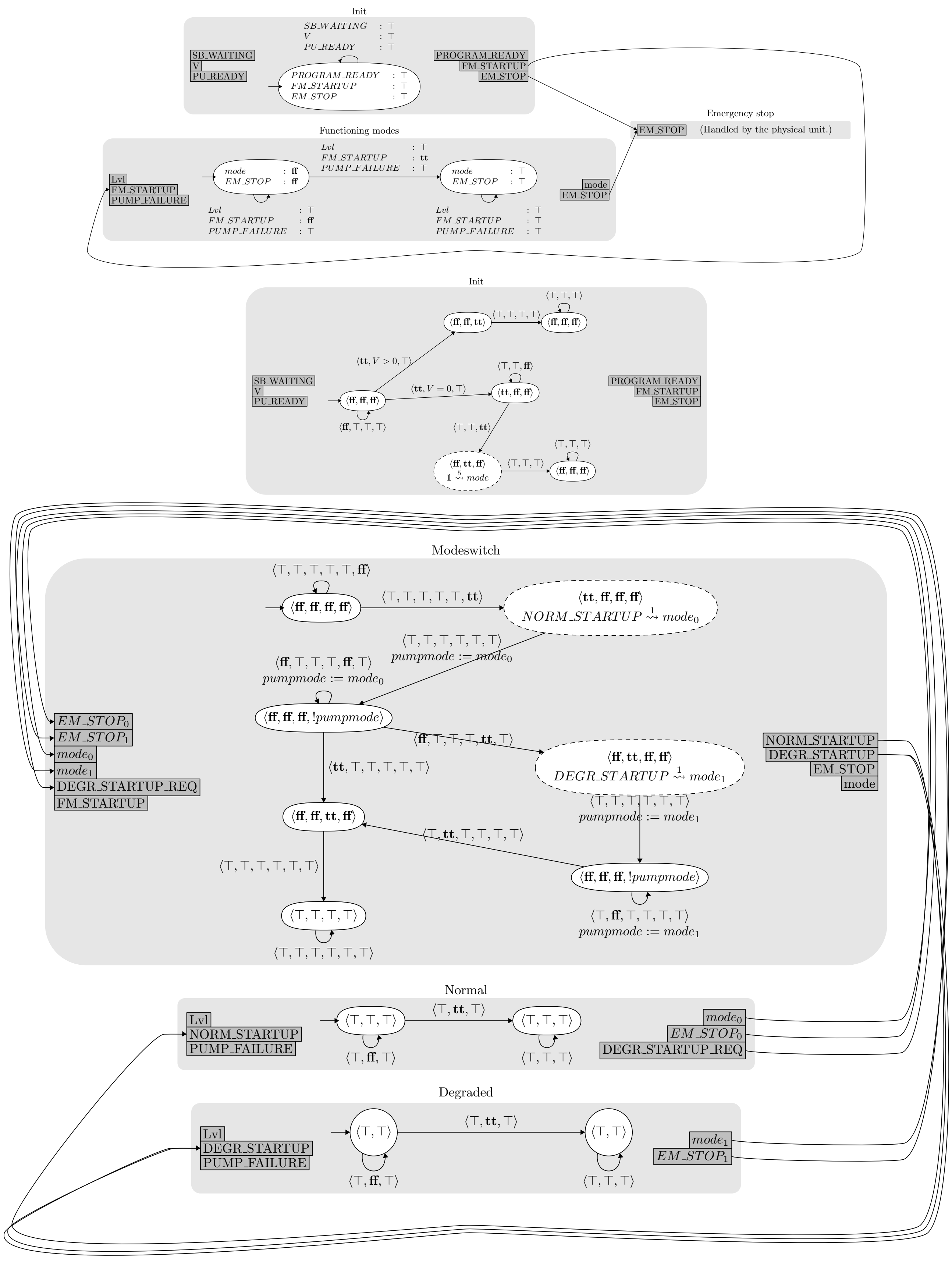


Figure 6: Refinement of INIT and FUNCTIONING MODES.

functioning mode. It then simply routes the data incoming on the $mode_0$ input port to the output $mode$ port. Whenever one of this data is a DEGR_STARTUP_REQ message, it starts up the DEGRADED machine and proceeds to routing its data. Starting the NORMAL and DEGRADED machines is submitted to an immediate reactivity constraint, which are not verified at this refinement level.

First refinement of NORMAL. The informal description of the NORMAL functioning mode specifies that the computation is to be done periodically. We peform a design choice by setting the period to 5 units of time. This directly gives us the global structure of the machine (cf. Fig. 7). After receiving a NORM_STARTUP message from MODESWITCH, the NORMAL machine enters a state in which it tests its inputs against still undefined conditions. These conditions can also entail the emission of an EM_STOP_0 or a $DEGR\_STARTUP\_REQ$ message. We specify that: on input-output ports Lvl of NORMAL to EM_STOP of MODESWITCH, state p_1 must have a reaction time less that 2; on input-output ports Lvl of NORMAL to $mode$ of MODESWITCH, if $PUMP\_FAILURE$ is false then p_1 must have a reaction time less that 5; on input-output ports Lvl of NORMAL to $DEGR\_STARTUP$ of MODESWITCH, if $PUMP\_FAILURE$ is true then p_1 must have a reaction time less that 5; from state q_{es}, there must be an immediate observable reaction in port EM_STOP of MODESWITCH; from state q_{deg}, there must be an immediate observable reaction in port $DEG\_STARTUP$ of MODESWITCH. None of these constraints are verified at this point of the refinement.

Last refinement of NORMAL. This last refinement enriches the preceding one by defining the control flow. The machine enters its control loop after receiving a NORM_STARTUP message. It always assumes a maximal behavior of the physical system. The quantity of water poured in n units of time while in NORMAL mode is $n.2.l_s$: as shown in Fig. 7, this is taken into account when testing Lvl. Moreover, when an message PUMP_FAILURE becomes visible and the integrity of the system is guaranteed, the NORMAL machine sends a request to activate the DEGRADED mode and goes into a non-reactive state. In order to illustrate our formalism, we will prove the partial correctness of this final refinement. The simulation relation is $\{(q_x, p_x) \text{ for all } x\}$. Let's consider $Lvl \overset{2}{\rightsquigarrow} EM\_STOP$: if $PUMP\_FAILURE$ = **ff** , we can exhibit transitions $q_1 \xrightarrow{\langle x,\top,\top\rangle} q_{es}$ and $q_1 \xrightarrow{\langle y,\top,\top\rangle} q_2$ s.t. $x + 2l_s \geq L_c$, $y + 2l_s < L_C$ and $\pi_{EM\_STOP_0}(\Gamma(q_{es})) \neq \pi_{EM\_STOP_0}(\Gamma(q_2))$. If $PUMP\_FAILURE$ = **tt** , we can exhibit transitions $q_1 \xrightarrow{\langle x,\top,\top\rangle} q_{es}$ and $q_1 \xrightarrow{\langle y',\top,\top\rangle} q_{deg}$ with similar properties. Hence, for all contexts, state q_1 is reactive from port Lvl to port EM_STOP_0 in time 0, i.e. with a separator of length 0. Moreover, state q_{es} verifies an immediate reactivity constraint as can be checked by performing the composition of the involved machines. This implies that state p_1 is reactive in one unit of time. The other constraints are verified similarly.

We have exposed the stepwise development of a steam-boiler control program. The three initial temporal constraints on reaction time and periodicity were decomposed into 8 constraints on reaction time, using four steps of refinement.

4 Conclusions and future works

We have described in this paper a formal framework dedicated to the time-based refinement of communicating systems modeled as Moore machines. The framework allows reasoning on the notion of *reaction time*, but is extensible to other metrics by associating to each transition an arbitrary cost, e.g. energy consumption. This allows to assess the cost of computation paths statically, which in turns may allow more precise schedules and more accurate and robust plans. Our refinement relation supports local modifications, allowing scalable verification and the independent development of modules. In order to illustrate our method, we also sketched an example which also emphasizes the applicability of our method to safety related real-time systems. Moreover, our method being based on finite state machines allows the application of all standard model-checking techniques to the verification of other properties.

An important observation we made is that functional dependencies are only weakly compositional (Garnier et al. 2011). Our framework is thus quite restrictive. Enriching specifications in order to make it more flexible would prove fruitful. Our work opens some other research directions: a broader investigation of the notion of reaction time in a more general setting (Haghverdi, Tabuada, and Pappas 2005) could prove fruitful and lead to simpler, more abstract and general definitions.

We provide a formal development framework based on time and associated automated verification tools which should be able to help the software designer and programmer to deliver reliable, predictable and efficient systems.

References

Abramsky, S. 1996. Retracing some paths in process algebra. In *CONCUR '96: Concurrency Theory, 7th International Conference*, 1–17. Springer-Verlag.

Abrial, J.-R. 1996. Steam-boiler control specification problem. *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler.*

Alur, R., and Dill, D. L. 1994. A theory of timed automata. *Theoretical Computer Science* 126:183–235.

Barbuti, R.; Bernardeschi, C.; and De Francesco, N. 2002. Abstract interpretation of operational semantics for secure information flow. *Inf. Process. Lett.* 83(2).

Bolognesi, T., and Brinksma, E. 1987. Introduction to the iso specification language lotos. *Computer Networks* 14:25–59.

Clarke, E.; Long, D.; and McMillan, K. 1989. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, 353–362. Piscataway, NJ, USA: IEEE Press.

Cousot, P., and Cousot, R. 1976. Static determination of dynamic properties of programs. In *Proceedings of the Sec-*

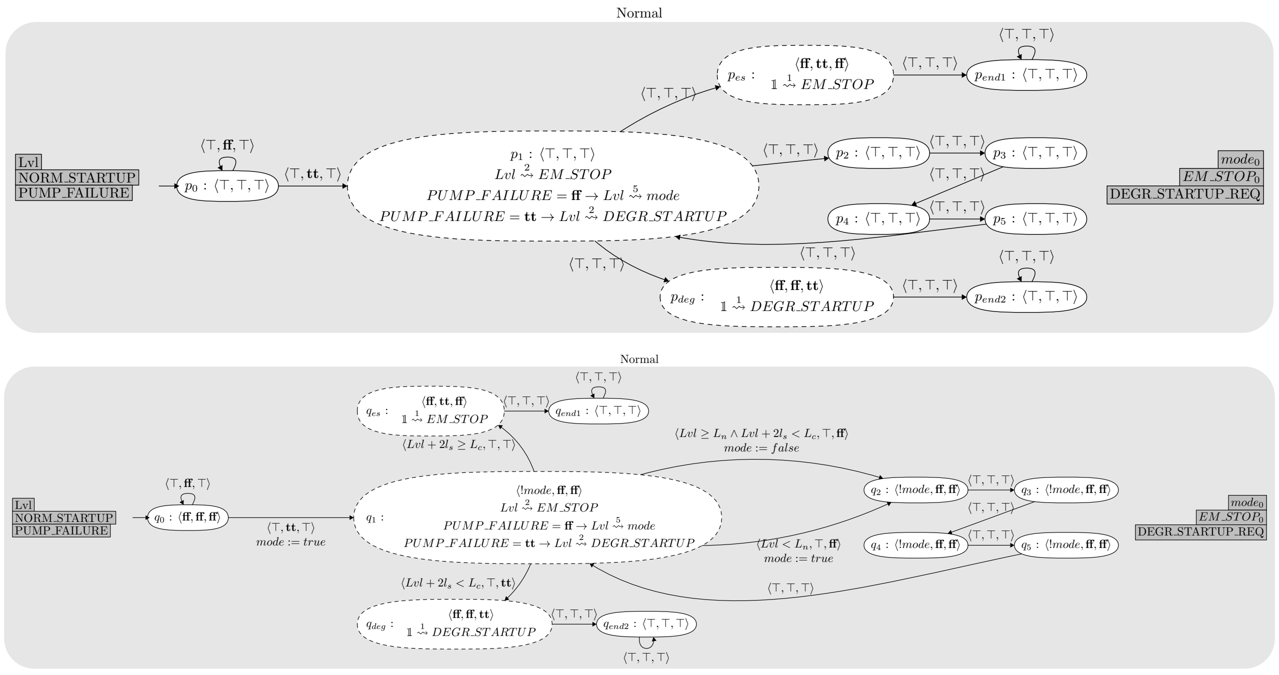

Normal
Lvl
NORM_STARTUP
PUMP_FAILURE
$p_0 : \langle\top,\top,\top\rangle$
$\langle\top,\mathbf{ff},\top\rangle$
$\langle\top,\mathbf{tt},\top\rangle$
$p_1 : \langle\top,\top,\top\rangle$
$Lvl \overset{2}{\rightsquigarrow} EM\_STOP$
$PUMP\_FAILURE = \mathbf{ff} \rightarrow Lvl \overset{5}{\rightsquigarrow} mode$
$PUMP\_FAILURE = \mathbf{tt} \rightarrow Lvl \overset{2}{\rightsquigarrow} DEGR\_STARTUP$
$p_{es} :$ $\langle\mathbf{ff},\mathbf{tt},\mathbf{ff}\rangle$ $\mathbb{1} \overset{1}{\rightsquigarrow} EM\_STOP$
$p_{end1} : \langle\top,\top,\top\rangle$
$p_2 : \langle\top,\top,\top\rangle$
$p_3 : \langle\top,\top,\top\rangle$
$p_4 : \langle\top,\top,\top\rangle$
$p_5 : \langle\top,\top,\top\rangle$
$p_{deg} :$ $\langle\mathbf{ff},\mathbf{ff},\mathbf{tt}\rangle$ $\mathbb{1} \overset{1}{\rightsquigarrow} DEGR\_STARTUP$
$p_{end2} : \langle\top,\top,\top\rangle$
$mode_0$
EM_STOP_0
DEGR_STARTUP_REQ
Normal
Lvl
NORM_STARTUP
PUMP_FAILURE
$q_0 : \langle\mathbf{ff},\mathbf{ff},\mathbf{ff}\rangle$
$\langle\top,\mathbf{ff},\top\rangle$
$\langle\top,\mathbf{tt},\top\rangle$ $mode := true$
$q_1 :$ $\langle !mode,\mathbf{ff},\mathbf{ff}\rangle$
$Lvl \overset{2}{\rightsquigarrow} EM\_STOP$
$PUMP\_FAILURE = \mathbf{ff} \rightarrow Lvl \overset{5}{\rightsquigarrow} mode$
$PUMP\_FAILURE = \mathbf{tt} \rightarrow Lvl \overset{2}{\rightsquigarrow} DEGR\_STARTUP$
$q_{es} :$ $\langle\mathbf{ff},\mathbf{tt},\mathbf{ff}\rangle$ $\mathbb{1} \overset{1}{\rightsquigarrow} EM\_STOP$
$\langle Lvl + 2l_s \geq L_c,\top,\top\rangle$
$q_{end1} : \langle\top,\top,\top\rangle$
$\langle Lvl \geq L_n \wedge Lvl + 2l_s < L_c,\top,\mathbf{ff}\rangle$ $mode := false$
$q_2 : \langle !mode,\mathbf{ff},\mathbf{ff}\rangle$
$q_3 : \langle !mode,\mathbf{ff},\mathbf{ff}\rangle$
$q_4 : \langle !mode,\mathbf{ff},\mathbf{ff}\rangle$
$q_5 : \langle !mode,\mathbf{ff},\mathbf{ff}\rangle$
$\langle Lvl < L_n,\top,\mathbf{ff}\rangle$ $mode := true$
$\langle Lvl + 2l_s < L_c,\top,\mathbf{tt}\rangle$
$q_{deg} :$ $\langle\mathbf{ff},\mathbf{ff},\mathbf{tt}\rangle$ $\mathbb{1} \overset{1}{\rightsquigarrow} DEGR\_STARTUP$
$q_{end2} : \langle\top,\top,\top\rangle$
$mode_0$
EM_STOP_0
DEGR_STARTUP_REQ

Figure 7: Refinement of NORMAL, first and second steps.

ond International Symposium on Programming, 106–130. Dunod, Paris, France.

Cousot, P., and Cousot, R. 1992. Inductive definitions, semantics and abstract interpretations. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 83–94. New York, NY, USA: ACM.

David, V.; Delcoigne, J.; Leret, E.; Ourghanlian, A.; Hilsenkopf, P.; and Paris, P. 1998. Safety properties ensured by the oasis model for safety critical real-time systems. In *SAFECOMP*.

David, A.; Larsen, K. G.; Legay, A.; Nyman, U.; and Wasowski, A. 2010. Timed i/o automata: a complete specification theory for real-time systems. In *Hybrid Systems*, 91–100.

Garnier, I.; Aussaguès, C.; David, V.; and Vidal-Naquet, G. 2011. On the reaction time of some synchronous systems. In *Proceedings of the 4th Interaction and Concurrency Experience (to appear)*.

Haghverdi, E.; Tabuada, P.; and Pappas, G. J. 2005. Bisimulation relations for dynamical, control, and hybrid systems. *Theor. Comput. Sci.* 342:229–261.

Meng, S., and Barbosa, L. S. 2006. Components As Coalgebras: The Refinement Dimension. *Theoretical Computer Science* 351:276 – 294.

Moore, E. F. 1956. Gedanken Experiments on Sequential Machines. In *Automata Studies*. Princeton U. 129–153.

Nielsen, M.; Havelund, K.; Wagner, K. R.; and George, C. 1988. The raise language, method and tools. In *VDM Europe*, 376–405.

Schmidt, D. A. 1995. Natural-semantics-based abstract interpretation (preliminary version). In *SAS '95: Proceedings of the Second International Symposium on Static Analysis*, 1–18. London, UK: Springer-Verlag.

Sifakis, J. 2009. Component-based construction of real-time systems in bip. In *CAV*, 33–34.

Winskel, G. 1984. Synchronization trees. *Theor. Comput. Sci.* 34:33–82.