Symbolic Execution Techniques Extended to Systems.

Christophe Gaston, Marc Aiguier, Diane Bahrami, Arnaud Lapitre

Symbolic Execution Techniques Extended to Systems

Christophe Gaston
CEA LIST, Boîte courrier 65,
F-91191 Gif sur Yvette Cedex
christophe.gaston@cea.fr

Marc Aiguier
Ecole Centrale de Paris
Laboratoire MAS
F-92295 Chatenay Malabry
marc.aiguier@ecp.fr

Diane Bahrami, Arnault Lapître
CEA LIST, Boîte courrier 65,
F-91191 Gif sur Yvette Cedex
diane.bahrami@cea.fr
arnault.lapitre@cea.fr

Abstract

This paper presents a symbolic execution framework devoted to system models, recursively defined by interconnecting component models. Our concern is to allow one to explicitly define interaction rules between components, while taking into account those rules at the symbolic execution phase. The paper introduces a small set of primitives dedicated to this purpose, together with their associated symbolic execution rules.

1 Introduction

Symbolic execution has been first defined for programs [12, 4]. It mainly consists in replacing concrete input values and initialization values of variables by symbolic ones in order to compute constraints induced on these variables by different possible executions. Symbolic execution has been widely used in different contexts to formally reason about programs. It provides all necessary pieces of information to detect unfeasible paths or deadlocks. Among its numerous applications, it has been used in combination with model checking technics to address verification and testing of programs [1, 11]. It has also been applied on models for verification or conformance testing purposes [14, 9, 5, 6].

Behavioral models of systems (as *StateCharts of Statemate* [10], *IF* [3], *UML statemachines* [13])...) are usually recursively defined from component models interacting accordingly to some rules that define communication or synchronization mechanisms between components. Those mechanisms depend on the considered modeling languages. Our contribution[1] is a first attempt to define a generic symbolic execution framework to take into account such system models. In our approach, system descriptions are given in the form of so called *designs* which introduce *programs* denoting system executions. A design may reuse other designs representing sub systems and in that case its program defines the interaction rules between reused designs. Designs may declare times (either discrete of dense) used to associate dates to observations of their executions and to define time constraints upon them. From a design point of view, its reused designs are black boxes: their executions are only observable by successive observations of values assigned to their *ports* which are used to exchange values. Such observations may be associated with dates and are symbolically represented as so called *snapshots*. The whole symbolic execution of a reused design is represented as a direct acyclic graph, called an *execution graph*, whose nodes are snapshots and whose arrows denote observational state evolutions during the reused design execution. Programs are built by structuring basic programs whose application is pre-conditioned by predicates over the system state. States of systems are symbolically denoted as so called *synchronizations* characterizing global observations of all reused designs (in the form of sets of reused design snapshots) and constraints over the main design port values and time. Basic programs may introduce instructions dedicated to model exchanges of values between different reused designs and thus represent communication rules. Basic program executions result in the definition of one or several new synchronizations. Basic programs are structured through several operators to schedule their executions. Structured program symbolic executions are built by successively executing basic programs accordingly to their scheduling. The symbolic execution of a design is then defined by extracting an execution graph from the symbolic execution of its program. This makes the symbolic execution procedure recursive that is, any design may be reused in some new design for which the symbolic execution procedure is applicable.

Denoting and symbolically executing models of in-

[1]This work has been partially supported by the $Usine\ Logicielle$ project of the french cluster $System@tic$ (Pôle de compétitivité) and by the french national ANR project $Hecosim$ (http://projet-hecosim.org/).

teraction has already been addressed at the requirement level [15, 16] by considering variants of message sequence charts as modeling languages. Our proposal is complementary because the models that we consider are component-oriented (designs are components that can be composed recursively). They are close to the models that can be described using formal languages as *CommUnity* [7] and *BIP* [2]. However, to the best of our knowledge, no works address the problem of symbolically executing system models defined from components interacting accordingly to explicitly specified interaction rules.

In Section 2 we introduce designs. Section 3 is devoted to symbolic execution rules. Section 4 is a discussion.

2 Design

Designs introduce data. In the following, such data are modeled by means of a data specification $SP = (\Omega, ax)$ as defined in the appendix and supposed given in the sequel. A *design* $\mathcal{D}$ is composed of a *public part*, a *private part* and a *body*.

design $design\_id$
public:
port $p_1[\textbf{:=}t_1] : s_1, \cdots, p_n[\textbf{:=}t_n] : s_n$
time $discrete|dense$
[**private:**
[**var** $v_1[\textbf{:=}t_1] : s_1, \cdots, v_m[\textbf{:=}t_m] : s_m$]
[**use** $design\_id_1, \cdots, design\_id_k$]]
body: $prog$

$design\_id$ is a *design identifier*. The *public* section introduces *typed ports* which are typed variables and a *time carrier* which is used to assign dates to observations of port values. That time carrier may be *discrete* or *dense*. Time is handled as the other data: we suppose that the set of types of Ω contains $Time_{di}$ (discrete) and $Time_{de}$ (dense), and that the set of operations contains at least, for each $i \in \{di, de\}$, an initial date denoted as a constant $0_i :\rightarrow Time_i$, and two function names $+_i : Time_i \times Time_i \rightarrow Time_i$ and $<_i$: $Time_i \times Time_i \rightarrow bool$ [2]. Semantical interpretations of those operations are the usual ones respectively over $\mathbb{N}$ and $\mathbb{Q}^+$. The *private* section introduces local typed variables used for computation and design identifiers denoting reused designs. All public variables involved in programs (ports and time) are prefixed by their design identifier. Private variables are not prefixed.

[2] We suppose that Ω also contains the sort *bool* with constants $true$ and $false$. Moreover, we will note $<$ instead of $<_i$ when the context is clear of confusion.

Programs introduced in the *body* section are defined as follows:

$$
\begin{array}{rcl}
Prog & ::= & \textbf{when}(For)\ \{Ins\}\ (For) \\
 & | & \{p \odot p'\} \text{ with } \odot \in \{\textbf{|;|}\ ,\ \textbf{|,|}\ ,\ \textbf{|i|}\} \\
 & | & \textbf{while}(For)\{Prog\} \\
Ins & ::= & \textbf{in}\ (design\_id.p)\ \textbf{provided}(For) \\
 & | & d.p\ \textbf{->}\ d'.p' \\
 & | & x\textbf{:=}t \\
 & | & \textbf{if}(For)\ \textbf{then}\{Ins\}[\textbf{else}\ \{Ins\}] \\
 & | & Ins; Ins
\end{array}
$$

where:

- x is a private variable or a port of $design\_id$.
  t is a term over Ω, over private and public variables of $\mathcal{D}$ and over public variables of its reused designs.
- $d \in \{design\_id, design\_id_1, \cdots, design\_id_k\}$, $d' \in \{design\_id_1, \cdots, design\_id_k\}$ and p and p' are respectively ports of d and d'.

$$
\begin{array}{rcl}
For & ::= & t_1 = t_2 \text{ where } t_1 \text{ and } t_2 \text{ are of same type} \\
 & | & \textbf{touch}(design\_id.p) \\
 & | & For \odot For \text{ with } \odot \in \{\wedge, \vee\} \\
 & | & \neg For
\end{array}
$$

Intuitively, $when(\varphi)\{ins\}(\psi)$ means: when φ is satisfied, the instruction ins is executed and after this execution the property ψ holds.

A design may receive a value to be assigned to its ports from its environment. Such an available value is finally assigned to the corresponding port by executing an instruction $in(design\_id.p)provided(\varphi)$ (occurring in the design program) provided that φ is evaluated to $true$. A design may also make some value available for one of its reused designs d' by means of the instruction $d.p$ **->** $d'.p'$ (d is either a reused design or is $design\_id$). Finally, $touch(design\_id.p)$ is a predicate evaluated to $true$ when a value is available for p.

Operators $|;|$ and $|i|$ are used to define the evaluation order of programs: $|;|$ is the sequencing of programs and $|i|$ is the interleaving operation which states that the order is arbitrary. $|,|$ is a choice operator.

Example 1 *Car Wiper Controller : design*

The system describes a simplified automatic car wiper controller. The main design ***env*** *represents the environment, which sends information about the intensity of the rain to* ***calc*** *every 5 time slot* **(1)**. ***calc*** *calculates the wipers' speed. When the calculated speed has changed in* ***calc*** **(3)**, ***env*** *sends it* **(4)** *from* ***calc*** *to the wiper* ***engine****, which only consumes the received value.*

```
    design env
public:
    port intensity := 0 : int,
    time discrete
private:
    var t := 0 : time, previousSpeed := 0 : int
    use calc, engine
body:
    while(True) {
    ( when (env.time = 5 + t) {                    (1)
        if ( touch(env.intensity) ){
          in(env.intensity);
          env.intensity -> calc.intensity
        }
        t := env.time
      }
  |,|
      when ( env.time < t + 5 &                    (2)
             calc.speed <> previousSpeed ) {       (3)
        previousSpeed := calc.speed;
        calc.speed -> engine.speed                 (4)
      }
   )
  }

    design calc
public:
    port intensity := 0 : int, speed := 0 : int
private:
    var th1 : int, th2 : int, speedTmp : int
body:
    while(True) {
      when (touch(calc.intensity) ) {
        in(calc.intensity);
        if(calc.intensity<th1)
        then{speedTmp := 1}
        if((calc.intensity>=th1)
            &
           (calc.intensity<th2) )
        then{speedTmp := 2}
        if(calc.intensity>=th2)
        then{speedTmp := 3}
        calc.speed:=speedTmp;
      }
    }

    design engine
public:
    port speed := 0 : int
body:
    while(True) {
      when (touch(engine.speed) ) {
        in(engine.speed);
      }
    }
```

The communication between ***env*** *and* ***calc*** *is time-triggered* **(1)** *since it occurs every 5 time slots (provided that a new intensity is received). On the contrary, the communication between* ***calc*** *and* ***engine*** *is event-triggered* **(3)***, since it occurs when a new speed is received. Constraint* **(2)** *ensures that those two communications occur at separate times.*

3 Symbolic execution

Symbolic executions are represented as graphs where nodes are symbolic system states and transitions are symbolic state evolutions. Symbolic states are characterized by symbolic values associated to variables and constraints on those values. In the sequel, symbolic values are denoted as terms over a set of fresh variables F disjoint from the set of variables of the program. In our context, states are (symbolic) snapshots in which each port is associated with both a current value (obtained via the function η in Def. 1), and an available (or buffer) value (via the function ν). Snapshots also introduce observation dates (δ) denoting values of their time carriers and sets of constraints π computed all along the execution. We note $P(\mathcal{D})$ the set of ports of a design $\mathcal{D}$[3] .

Definition 1 (Snapshots) *A* snapshot *over $\mathcal{D}$ is any element (η, ν, δ, π) of $(T_\Omega(F))^{P(\mathcal{D})} \times (T_\Omega(F) \cup \{\bot\})^{P(\mathcal{D})} \times T_\Omega(F)_{time_i} \times 2^{Sen_\Omega(F)}$ such that π is finite. We note $Snp(\mathcal{D})$ the set of all snapshots over $\mathcal{D}$.*

$\bot$ is used to model the absence of available values. In the sequel, for any such snapshot s, the notations η_s, ν_s, δ_s and π_s stand respectively for η, ν, δ and π.

Example 2 *Car Wiper Controler : snapshot*
Here is a snapshot for design ***env*** *at initialization time:*

$$\begin{pmatrix} \eta_0(intensity) & \to i_0 \\ \nu_0(intensity) & \to \bot \\ t_0 & \\ \pi_0\{t_0 \geq 0,\ i_0 = 0\} & \end{pmatrix}$$

The current value of port intensity is the symbolic value i_0, constrained to 0, and there are no new values available for this port.

[3] In the sequel, for any two sets A and B, B^A denotes the set of all functions from A to B.

The symbolic execution of a design $\mathcal{D}$ is given in the form of a particular structure, the so-called *execution graph* of $\mathcal{D}$.

Definition 2 (Execution graph) *An* execution graph *of $\mathcal{D}$ is a couple (I, R) where $I \in Snp(\mathcal{D})$ and $R \subseteq Snp(\mathcal{D}) \times Snp(\mathcal{D})$ are such that the following conditions hold:*

(Initialization). *η_{Init} is injective and*
$\forall x \in P(\mathcal{D}), \eta_{Init}(x) \in F \wedge \nu_{Init}(x) = \bot$

(Time). $\forall (s_1, s_2) \in R$, $\delta_{s_2} > \delta_{s_1} \in \pi_{s_2}$

Initialization ensures that, at the initial snapshot, ports are assigned by variables of F and no values are available from the environment yet. *Time* ensures that successive observations occur consistently with time passing.

The symbolic execution of a design is built by symbolically executing its associated program. Some operations and conditions occurring in programs refer to states of reused designs. Therefore, states of reused designs have to be known. Thus, a complete state for a design is given by one of its own snapshots together with snapshots for its reused designs. Such a complete state observation is called a *synchronization*. In the sequel, $V(\mathcal{D})$ refers to the set of private variables declared in $\mathcal{D}$.

Definition 3 (Synchronization) *Let $\mathcal{D}_1, \cdots \mathcal{D}_k$ be the reused designs of $\mathcal{D}$. A* $\mathcal{D}$-synchronization *is a triple $sync = (s, i, \gamma)$ where $s \in Snp(\mathcal{D})$, $i : V(\mathcal{D}) \rightarrow T_\Omega(F)$ is a mapping, and $\gamma \in \prod_{j \leq k} Snp(\mathcal{D}_j)$.*

We note γ_j for the j^th-projection of γ and $Sync(\mathcal{D})$ the set of all $\mathcal{D}$-synchronizations.

Note that beside snapshots, synchronizations also introduce values associated to private variables (function i) which do not appear in snapshots since they are not observable from the design environment. One can define a unique substitution associated to $sync$, denoted ι_{sync}, which relates variables of $sync$ to its symbolic values. More precisely ι_{sync} associates: to any $x \in P(\mathcal{D})$ (respectively $x \in P(\mathcal{D}_j)$ for some $j \leq k$) the term $\eta_s(x)$ (respectively $\eta_{\gamma_j}(x)$); to any $x \in V(\mathcal{D})$ the term $i(x)$; to $design\_id.time$ (respectively, to $design\_id_j.time$ with $j \leq k$) the variable δ_s (respectively δ_{γ_j}).

We also note ι_{sync} the canonical extension to formulas which associates to any formula φ the formula $\iota_{sync}(\varphi)$ obtained by replacing: (1) occurrences of any ports or variables x by $\iota_{sync}(x)$; (2) sub-formulas of the form $touch(design\_id.x)$ (resp. $touch(design\_id_j.x)$ with $j \leq k$) by $True$ if $\nu_s(design\_id.x) \neq \bot$ (resp. $\nu_{\gamma_j}(design\_id_j.x)) \neq \bot$) and $False$ otherwise. Let us recall that $touch(x)$ is true in a given snapshot s when a value is available for x in s, that is $\nu_s(x) \neq \bot$.

We now define *symbolic execution of instructions* as mathematical relations between synchronizations (intuitively $sync$ is related to $sync'$ if $sync'$ reflects the effect of an instruction execution from $sync$). The execution of an instruction may affect reused designs. The resulting effect on reused design snapshots must be compatible with the possible executions of that reused designs. Those possible executions of reused designs are represented by a collection of (previously computed) execution graphs associated to them.

Definition 4 (Symbolic execution of instructions) *For $j \leq k$, let us note $\mathcal{G}_j$ an execution graph of the design $\mathcal{D}_j$ and $\mathcal{G} = (\mathcal{G})_{j \leq k}$. For any instruction ϱ, let us note $[\![\varrho]\!]_\mathcal{G} \subseteq Sync(\mathcal{D}) \times Sync(\mathcal{D})$ the* symbolic execution of *ϱ inductively defined on the structure of instructions (s, i, γ) $[\![\varrho]\!]_\mathcal{G}$ (s', i', γ') if, and only if $\delta_{s'} = \delta_s$, and:*

- *if $\varrho \equiv x := t$ then $\gamma' = \gamma$ and:*
  - *if $x \in P(\mathcal{D})$, then*[4]*, $\eta_{s'} = \eta_{s_{x \mapsto a}}$ for some fresh variable $a \in F$, $\nu_{s'} = \nu_s$, $i' = i$ and $\pi_{s'} = \pi_s \cup \{a = \iota_{(s,i,\gamma)}(t)\}$*
  - *if $x \in V(\mathcal{D})$ then $i' = i_{x \mapsto a}$ for some fresh variable $a \in F$, $\pi_{s'} = \pi_s \cup \{a = \iota_{(s,i,\gamma)}(t)\}$, $\eta_{s'} = \eta_s$ and $\nu_{s'} = \nu_s$*
- *if $\varrho \equiv in(x)provided(\varphi)$, then:*
  $\nu_s(x) \neq \bot$, $\gamma' = \gamma$,
  $\eta_{s'} = \eta_{s_{x \mapsto a}}$ for some $a \in F$, $\nu_{s'} = \nu_{s_{x \mapsto \bot}}$ and[5]
  $\pi_{s'} = \pi_s \cup \{[\iota_{(s,i,\gamma)}(\varphi)]_{x \leftarrow a} \wedge a = \nu_s(x)\}$,
- *if $\varrho \equiv x$->$design\_id_j.p'$, then:*
  - *$\nu_{\gamma_j}(design\_id_j.p') = \bot$,*
  - *for all $l \neq j \leq k$ $\gamma'_l = \gamma_l$,*
  - *if we note $\mathcal{G}_j = (I_j, R_j)$ then $\gamma_j \; R_j \; \gamma'_j$,*
  - *$\eta_{\gamma'_j} = \eta_{\gamma_j}$, $\nu_{\gamma'_j} = \nu_{\gamma_j \, design\_id_j.p' \mapsto a}$, and*
  - *s' is similar to s except that $\pi_{s'}$ is $\pi_s \cup \{a = \iota_{(s,i,\gamma)}(x)\}$*
- *sequences and conditions are handled as usual.*

[4] For any function $f : A \rightarrow B$, $x \in A$ and $y \in B$, $f_{x \mapsto y} : A \rightarrow B$ is the function equal to f except it associates y to x.

[5] For any formula φ, $\varphi_{z \leftarrow y}$ is the formula φ where all occurrences of z are replaced by y.

$in(x)provided(\varphi)$ can not be executed from a synchronization where no values are available for x ; otherwise, it results in a synchronization in which x is assigned by the available value. $x \rightarrow design\_id.p'$ can not be executed from a synchronization where a value is already available for p' ; otherwise, it yields synchronizations where the snapshot corresponding to $\mathcal{D}_j$ is reachable from γ_j in the execution graph $\mathcal{G}_j$ and reflects that a new value (symbolically denoted by a) is available for x. Finally, s' is similar to s except that the constraint $a = \nu_{sync}(x)$ is added to reflect that the available value is the one assigned to x in $sync$.

Definition 4 defines how the state of a design can be modified by an instruction. But it can also be modified without executing any instructions. A design state (*i.e.* a snapshot) may evolve because either time passes or new values become available for some of its ports. To capture this fact we define the notion of *stuttering of a snapshot*.

Definition 5 (Stuttering) *The* stuttering of a snapshot s *is the set* $St(s)$ *of snapshots* s' *such that* $\eta_{s'} = \eta_s$*; for all* x*, if* $\nu_s(x) \neq \perp$ *then* $\nu_{s'}(x) = \nu_s(x)$*, otherwise* $\nu_{s'}(x) = \perp$ *or is a fresh variable;* $\delta_{s'} = a$ *and* $\pi_{s'} = \pi_s \cup \{a > \delta_s\}$ *with* $a \in F$*.*

Snapshots of reused designs may also evolve without being controlled by the main design program: this comes from internal executions of reused design programs (*i.e.* executions of reused designs that do not require the availability of new values on their ports). Snapshots resulting from such executions can be identified in reused design execution graphs.

Definition 6 (Internally reachable snapshots) *Let* s *be a snapshot of some execution graph* $G = (Init, R)$*. The* set of snapshots internally reachable from s in G*, denoted* $\mathcal{I}^*_G \subseteq Snp(\mathcal{D}) \times Snp(\mathcal{D})$ *is the transitive closure of:*

$$(s\mathcal{I}_G s') \Leftrightarrow \begin{cases} s \; R \; s' \\ \forall x \in P(\mathcal{D}), \nu_{s'}(x) \neq \perp \Rightarrow \\ \qquad\qquad \nu_{s'}(x) = \nu_s(x) \end{cases}$$

s' is internally reachable from s when it is reachable through R^* (the transitive closure of R) and none of the ports has received a new available value. When $\nu_s(x) \neq \perp$ and $\nu_{s'}(x) = \perp$ this reflects a consumption of the value available for x.

We now define sets of so called *uncontrollable futures of a synchronization*, which denote state evolutions corresponding to stuttering for the main design and internal executions for reused designs.

Definition 7 (Uncontrollable futures) *Let* $sync = (s, i, \gamma)$ *be a* $\mathcal{D}$*-synchronization and for every* $j \leq k$ *let* $\mathcal{G}_j$ *be an execution graph of* $\mathcal{D}_j$*. The* Uncontrollable futures of $sync$*, denoted* $\mathcal{F}(sync)$ *is the set of all* $\mathcal{D}$*-synchronizations* $sync' = (s', i, \gamma')$ *s.t.* $s' \in St(s)$ *and for all* $j \leq k$*,* $\gamma_j \; \mathcal{I}^*_{\mathcal{G}_j} \gamma'_j$*.*

We then define the *program symbolic execution*. The symbolic execution of a program relies on the symbolic executions of instructions introduced in that program. Given a set of execution graphs $\mathcal{G} = (\mathcal{G}_j)_{j \leq k}$, the symbolic execution of instructions is a binary relation on synchronizations. Therefore, the symbolic execution of programs is of the same form[6].

Definition 8 (Symbolic execution of programs) *With notations of Definition 4, the* symbolic execution of any program α *from a set of synchronizations* $Syn \subseteq Sync(\mathcal{D})$ *is the relation* $[\![\alpha]\!]^{Syn}_{\mathcal{G}} \subseteq Sync(\mathcal{D}) \times Sync(\mathcal{D})$ *defined as follows:*

- $[\![when(\varphi)\{\rho\}(\psi)]\!]^{Syn}_{\mathcal{G}}$ *is the set of all* $(sync, sync')$ *such that* $sync \in Syn$ *and if we note* $sync_1$ *the synchronization* (s_1, i_1, γ_1) *defined as* $sync = (s, i, \gamma)$ *except that* $\pi_{s_1} = \pi_s \cup \{\iota_{sync}(\varphi)\}$*, then*
  - *there exists a synchronization* $sync_2$ *such that* $sync_1 \; [\![\rho]\!]_{\mathcal{G}} \; sync_2$*, and*
  - *if we note* $sync_3$ *the synchronization defined as* $sync_2$ *except that* $\pi(s_3) = \pi(s_2) \cup \{\iota_{sync_2}(\psi)\} \bigcup_{j \leq k} \{\iota_{sync_2}(\pi_{\gamma_j})\}$*, then* $sync' \in \mathcal{F}(sync_3)$
- $[\![\alpha_1|;|\alpha_2]\!]^{Syn}_{\mathcal{G}} = [\![\alpha_1]\!]^{Syn}_{\mathcal{G}} \cup [\![\alpha_2]\!]^{Syn'}_{\mathcal{G}}$*, where* $Syn' = L([\![\alpha_1]\!]^{Syn}_{\mathcal{G}})$*,*
- $[\![\alpha_1|,|\alpha_2]\!]^{Syn}_{\mathcal{G}} = [\![\alpha_1]\!]^{Syn}_{\mathcal{G}} \cup [\![\alpha_2]\!]^{Syn}_{\mathcal{G}}$*,*
- $[\![\alpha_1|i|\alpha_2]\!]^{Syn}_{\mathcal{G}} = [\![\alpha_1|;|\alpha_2]\!]^{Syn}_{\mathcal{G}} \cup [\![\alpha_2|;|\alpha_1]\!]^{Syn}_{\mathcal{G}}$*,*
- *Programs of the form* $while(\varphi)\{\alpha\}$ *are handled as usual.*

We define the symbolic execution graph of a design by symbolically executing its associated program from an initial synchronization denoting an observation of the system at the initial state and by forgetting snapshots of reused designs.

[6] In the sequel, for any set E and relation $R \subseteq E \times E$ we will note $L[R]$ the set of *leaves of* R defined as $\{y \in E | \exists x \in E. x \; R \; y \wedge \forall z \in E. \neg(y \; R \; z)\}$.

Definition 9 (Symbolic execution of designs) *Let $\mathcal{D}$ be a design, $\mathcal{D}_1, \ldots, \mathcal{D}_k$ be its reused designs, and α be the program of $\mathcal{D}$. Let $\mathcal{G}(\mathcal{D}_1), \ldots, \mathcal{G}(\mathcal{D}_k)$ be already computed execution graphs for each sub-design $\mathcal{D}_1, \ldots, \mathcal{D}_k$. Let $sync = (s, i, \gamma)$ be an initial synchronization that satisfies:*

- *s verifies* **(Initialization)** *condition of Definition 2, and*
- *for every $x \in V(\mathcal{D})$, $i(x)$ is a fresh variable of F disjoint of any fresh variable that occurs in some $\mathcal{G}_i$ for $i = 1, \ldots, k$. Moreover, i is injective (i.e. two variables $x, y \in V(\mathcal{D})$ cannot be associated to the same fresh variable).*

Then, the symbolic execution *of $\mathcal{D}$ is the couple $\mathcal{G}(\mathcal{D}) = (s, R)$ where R is the set of couples of snapshots (s_1, s_2) for which there exist both $i_1, i_2 : V(\mathcal{D}) \to T_\Omega(F)$ and $\gamma_1, \gamma_2 \in \prod_{j \leq k} Snp(\mathcal{D}_j)$ such that $(s_1, i_1, \gamma_1)\ [\![\alpha]\!]_{\mathcal{G}}^{\{sync\}}\ (s_2, i_2, \gamma_2)$.*

Example 3 *Car Wiper Controller : symbolic execution*

The diagram below represents a part of the symbolic execution of env's program (from Ex. 1). Snapshots concerning the engine are not represented due to the lack of space. By only keeping snapshots concerning env (left column of the diagram) the resulting path belongs to the symbolic execution of env in the sense of Definition 9.

[env]

$$\left(\begin{array}{ll} \eta_0(intensity) & \to i_0 \\ \nu_0(intensity) & \to \bot \\ t_0 & \\ \pi_0 = \{t_0 \geq 0,\ i_0 = 0\} & \end{array}\right)$$

[calc]

$$\left(\begin{array}{ll} \eta'_0(intensity) & \to i'_0 \\ \eta'_0(speed) & \to s_0 \\ \nu'_0(intensity) & \to \bot \\ t'_0 & \\ \pi'_0 = \{t'_0 \geq 0,\ i'_0 = 0, & \\ \quad s_0 = 0\} & \end{array}\right)$$

$\Downarrow$ *new value available for port $env.intensity$*

$$\left(\begin{array}{ll} \eta_1(intensity) & \to i_0 \\ \nu_1(intensity) & \to i_1 \\ t_1 & \\ \pi_1 = \pi_0 \cup \{t_1 > t_0\} & \end{array}\right)$$

$\Downarrow$ *value of $env.intensity$ sent to $calc$*

$$\left(\begin{array}{ll} \eta_2(intensity) & \to i_1 \\ \nu_2(intensity) & \to \bot \\ t_2 & \\ \pi_2 = \pi_1 \cup \{t_2 > t_1, & \\ \quad t_2 = 5, & \\ \nu_2(calc.intensity) = i_1\} & \end{array}\right) \qquad \left(\begin{array}{ll} \eta'_1(intensity) & \to i'_0 \\ \eta'_1(speed) & \to s_0 \\ \nu'_1(intensity) & \to i'_1 \\ t'_1 & \\ \pi'_1 = \pi'_0 \cup \{t'_1 > t'_0\} & \end{array}\right)$$

$\Downarrow$ *$calc$ runs its code and computes $calc.speed$*

$$\left(\begin{array}{ll} \eta'_2(intensity) & \to i'_1 \\ \eta'_2(speed) & \to s_1 \\ \nu'_2(intensity) & \to \bot \\ t'_2 & \\ \pi'_2 = \pi'_1 \cup \{t'_2 > t'_1\} & \end{array}\right)$$

$\Downarrow$ *$calc.speed$ sent to $engine.speed$*

$$\left(\begin{array}{ll} \eta_3(intensity) & \to i_1 \\ \nu_3(intensity) & \to i_2 \\ t_3 & \\ \pi_3 = \pi_2 \cup \{t_3 > t_2,\ t_3 <> t_2 + 5, & \\ \eta_3(calc.speed) <> previousSpeed, & \\ \nu_3(engine.speed) = \eta_3(calc.speed)\} & \end{array}\right)$$

Theorem 1 *For any design $\mathcal{D}$:*

- *$\mathcal{G}(\mathcal{D})$ is an execution graph,*
- *$\mathcal{G}(\mathcal{D})$ is computable.*

Sketch of proof
The first item of Theorem 1 holds because **(Initialization)** is ensured by Definition 9 and **(Time)** is ensured by Definitions 7 and 8. That property ensures the ability to recursively and symbolically execute designs built in a hierarchical manner. The second item holds because the relation $[\![\alpha]\!]_{\mathcal{G}}^{Syn}$ from Definition 8 is defined by induction on the form of design programs.

4 Discussion

Of course the set of snapshots related by R may be infinite, when a design represents a reactive system (i.e. a system continuously interacting with its environment). In such a case, paths (*i.e.* executions) starting at the initial snapshot are arbitrary long. However Definition 9 can be associated with an algorithm which computes arbitrary long paths. Such an algorithm is sufficient for simulation or testing purposes. Each path of a symbolic execution characterizes in fact a class of concrete behaviors. Such behaviors are called *numerical executions*. Numerical executions are sequences of so-called *numerical snapshots*. Numerical snapshots are defined up to a model $M \in Mod(SP)$ and are triples $s = (\eta_s, \nu_s, \delta_s) \in of M^{P(\mathcal{D})} \times (M \cup \{\bot\})^{P(\mathcal{D})} \times (M_{time_i} \cup \{\varepsilon\})$. A numerical execution $p = s_1 \cdots s_n$ corresponds intuitively to a sequence of observations of observable states of $\mathcal{D}$ (*i.e.* numerical snapshots) during an execution. Between two observations, an input value may be sent from the environment of $\mathcal{D}$. Such an

input $v \in M$ for a port x occurs between two consecutive snapshots s and s' of p if and only if $\nu_s(p) = \perp$ and $\nu_{s'}(p) = v$. With notations of Definition 9, p is a numerical execution if there exists a sequence of snapshots $s'_1 \cdots s'_n$ with $s'_1 = s$ and $\forall k < n\ (s'_k, s'_{k+1}) \in R$ satisfying:

- $\exists i : F \to M$ such that $M \models_i \bigwedge_{\varphi \in \pi_{s'_n}} \varphi$ and,
- $\forall k \leq n, \forall x \in P(\mathcal{D})$,
  $\eta_{s_k}(x) = i(\eta_{s'_k}(x))$, $\nu_{s_k}(x) = i(\nu_{s'_k}(x))$ when $\nu_{s'_k}(x) \neq \perp$ and $\nu_{s_k}(x) = \perp$ otherwise,
  and finally $\delta_{s_k} = i(\delta_{s'_k})$.

5 Conclusion

We have proposed a framework to symbolically execute system models defined from components interacting accordingly to explicitly specified interaction rules. Systems are modeled by means of the notion of design. Designs may reuse other designs denoting sub systems. Designs introduce programs that allow one to specify executions in an imperative style. Those executions may contain value passing and synchronization between reused designs. Time constraints may be expressed and are taken into account symbolically.

Although we did not present it due to lack of space, designs have been associated with a formal semantics ([8]) which allows one to mathematically ground the symbolic execution rules that we introduce. The symbolic execution mechanisms are being currently implemented in the tool set $AGATHA$ [14]. This symbolic execution tool is associated to different rewriting tools[7] and sat-solvers[8] in order to treat various data types. The small set of primitives introduced in this paper is not sufficient to represent all the semantical features of models written using a real modelling formalism (for example dynamic creation of process or inheritance can not be captured). This set of primitives should be extended to take into account data structure and modelling mechanisms of each involved modelling language. However, a lot of work as already been done on this aspect in the AGATHA tool (AGATHA is already able to symbolically execute StateCharts of Statemate, IF models and UML statemachines). At the implementation level, our primitives come as an extension of the already defined ones. The notion of design has been implemented as a profile of UML in the Papyrus tool set[9]. Future works include experimentations with the AGATHA tool and connections with the UML profile. At a more theoretical level, semantics of designs are based on the idea that it is possible to build synchronizations which involve observations of all reused designs. Therefore, even though our framework allows one to deal with systems which execute asynchronously, we are not able yet to deal with distributed systems for which such synchronizations may not make sense. We are currently working on this issue.

[7] CafeOBJ, MAUDE: http://www.cs.ucsd.edu/ goguen/sys/obj.html.
[8] CVC3 http://www.cs.nyu.edu/acsys/cvc3/
[9] http://www.papyrusuml.org

References

[1] S. Anand, C. Pasareanu, and W. Visser. Symbolic execution with abstraction. *International Journal on Software Tools for Technology Transfer*, 11(1):53–67, 2009.

[2] A. Basu, M. Bozga, and J. Sifakis. Modeling Heterogeneous Real-time Components in BIP. In *SEFM*, pages 3–12, 2006.

[3] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis. The IF toolset. In *SFM*, pages 237–267, 2004.

[4] L.-A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on software engineering*, 2(3):215–222, September 1976.

[5] A. Faivre, C. Gaston, and P. Le Gall. Symbolic Model Based Testing for Component Oriented Systems. In Springer Berlin / Heidelberg, editor, *Testing of Software and Communicating Systems TestCom/FATES 2007*, volume 4581 of *LNCS*, pages 90–106, 2007.

[6] A. Faivre, C. Gaston, P. Le Gall, and A. Touil. Test Purpose Concretization through Symbolic Action Refinement. In Springer Berlin / Heidelberg, editor, *Testing of Software and Communicating Systems TestCom/FATES 2008*, volume 5047 of *LNCS*, pages 184–199, 2008.

[7] J. L. Fiadeiro, A. Lopes, and M. Wermelinger. A Mathematical Semantics for Architectural Connectors. volume 2793 of *LNCS*, pages 190–234. Springer-Verlag, 2003.

[8] C. Gaston, M. Aiguier, A. Lapitre, A. Cuccuru, and C. Mraidha. LEM : a language for expressing execution and interaction semantics. http://www.epigenomique.genopole.fr/˜aiguier, 2008.

[9] C. Gaston, P. Le Gall, N. Rapin, and A. Touil. Symbolic Execution Techniques for Test Purpose Definition. In *Testing of Communicating Systems: 18th IFIP TC 6/WG 6.1 International Conference, TestCom 2006. LNCS*, NY, USA, May 16-18 2006. Springer.

[10] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5:54–64, 1996.

[11] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568. Springer, 2003.

[12] J.-C. King. A new approach to program testing. *Proc. of the international conference on Reliable software, Los Angeles, California*, 21-23:228–233, April 1975.

[13] Object Management Group. Unified Modeling Language Specification, Version 1.5, 2003. www.omg.org/technology/documents/formal/uml.htm.

[14] N. Rapin, C. Gaston, A. Lapitre, and J.-P. Gallois. Behavioural unfolding of formal specifications based on communicating automata. In *Proc. of first Workshop on Automated technology for verification and analysis*, Taiwan, 2003.

[15] A. Roychoudhury, A. Goel, and B. Sengupta. Symbolic message sequence charts. In *ESEC-FSE '07: Proc. of the 6th joint meeting of the European software engineering conference and ACM SIGSOFT symposium on the foundations of software engineering*, pages 275–284, NY, USA, 2007. ACM.

[16] T. Wang, A. Roychoudhury, R. H. C. Yap, and S. C. Choudhary. Symbolic execution of behavioral requirements. In *PADL*, pages 178–192, 2004.

Data type formalism

A *data signature* is a couple $\Omega = (S, Op)$ where S is a set of types, Op is a set of operations, each one being provided with a profile $s_1 \cdots s_{n-1} \rightarrow s_n$ (for $i \leq n$, $s_i \in S$). A set of typed variables V is a set whose each element x is associated with a type by means of a function $type : V \rightarrow S$. The set $T_\Omega(V)$ of Ω-*terms* with typed variables in V is inductively defined as usual over Op and V. Terms of $T_\Omega(V)$ are associated to types in S by canonically extending the function $type$ to terms.

An Ω-*substitution* over V is a function $\sigma : V \rightarrow T_\Omega(V)$ preserving types which can also be canonically extended to $T_\Omega(V)$. $T_\Omega(V)^V$ denotes[10] the set of all Ω-substitutions defined on V. The set $Sen_\Omega(V)$ of all Ω-*typed equational formulas* contains the truth values *True* and *False* and all formulas built using the equality predicates $t = t'$ for $t, t' \in T_\Omega(V)$ s.t. $type(t) = type(t')$, the usual connectives $\neg$, $\vee$, $\wedge$ and quantifiers $\forall$, $\exists$. A *many sorted equational specification* is a couple $SP = (\Omega, ax)$ where Ω is a data signature and $ax \subseteq Sen_\Omega(V)$ is a finite set.

An Ω-*model* is a family $M = \{M_s\}_{s \in S}$ with, for each $f : s_1 \cdots s_n \rightarrow s \in Op$, a function $f_M : M_{s_1} \times \cdots \times M_{s_n} \rightarrow M_s$. M_s is called the *carrier of* s. Ω-*interpretations over* V are applications i from V to M preserving types, extended to terms in $T_\Omega(V)$. A model M satisfies a formula φ, denoted $M \models \varphi$, iff for all interpretations i, $M \models_i \varphi$, where $M \models_i t = t'$ is defined by $i(t) = i(t')$, and where the truth values and the connectives are handled as usual. M^V is the set of all interpretations from V to M. Semantics of SP, denoted $Mod(SP)$ are the set of all models M s.t. $M \models \varphi$ for all $\varphi \in ax$.

Each time the context is clear of confusion, the prefix "Ω" is removed (*terms* will be used instead of Ω-terms for instance).

[10] In the sequel, for any to sets A and B, B^A denotes the set of all functions from A to B.