Symbolic Execution and Heterogeneous Modeling

Christophe Gaston  
CEA LIST Saclay  
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

Arnault Lapitre  
CEA LIST Saclay  
F-91191 Gif sur Yvette Cedex  
arnault.lapitre@cea.fr

Abstract

This paper is devoted to complex system modeling, that is to say modeling recursively systems by interconnecting component models. Our concern is to address the potential heterogeneity induced by the usage of different modeling formalisms for describing different component models involved in a common system. The main difficulty is each modeling language has its own execution and communication semantics (e.g. synchronous, asynchronous, etc.). Therefore, modeling systems in the whole comes to model the way heterogeneous components interact. We propose a formal framework composed of: (a) a design language allowing to capture usual modeling language semantics and to describe rich interaction mechanisms; (b) symbolic execution mechanisms allowing to simulate system model executions for models described in our language. The complete framework allows to symbolically execute heterogeneous system models for formal validation purposes. Symbolic execution mechanisms associate de facto an operational semantics to our language.

1 Introduction

Modern systems are often complex, that is they are obtained by assembling components that interact according to communication policies. Moreover, modern systems are often heterogeneous. Typical examples are embedded systems which contain both software and hardware parts to pilot mechanical ones. The heterogeneity of realization has its counterpart at the modeling level since modeling components of different natures is more convenient if using different modeling languages specifically adapted to each model nature. For instance, UML [11] is adapted to model object-oriented pieces of software, VHDL − AMS [9] is more convenient for hardware parts ... However, a collection of heterogeneous models of components constituting a system does not form a model of the whole system: the way subcomponents interact should also be modeled. The main difficulty is that each subcomponent inherits semantics of execution and communication of its corresponding modeling language. Therefore modeling interactions consists in connecting different subcomponents behaviors described in different languages, in order to define resulting system behaviors. We propose1 to define a formal framework allowing to express and execute such heterogeneous models. More precisely, we define a simulation framework based on symbolic execution mechanisms to deal with heterogeneous systems. Symbolic execution has been first defined for programs [10, 2] and 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 the execution of the program. Symbolic execution allows to represent possible executions of a program in a very generic manner and provides all necessary pieces of information to detect unfeasible paths or deadlocks. We have already defined extension of symbolic execution to a communicating automata based formalism allowing to model reactive systems [12], detect deadlocks and we have shown how to use that technique for different purposes in conformance testing activities [7, 4, 5]. However our automata based formalism has its own, implicit execution and interaction semantics. In the current contribution we present a very generic language LEM (Language for Expressing Models of execution and interaction semantics) allowing to capture a wide class modeling language semantics. LEM is based on the notion of so-called design. Designs are used both to encode semantics of subcomponents in a format that can be taken into account by the symbolic execution engine and to describe interaction semantics between subcomponents.

1This 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/).
Related works. In [8, 3], simulation plate-forms for heterogeneous systems are proposed. Associated modeling frameworks allow one to hierarchically design models of systems in which each layer component is similar to our notion of design. However in both cases semantics of components and interactions between them are at some point defined by their implementations. Implementations are finally written in programming languages that do not yield a formal meaning. Our language $LEM$ could be used as a dedicated implementation language in order to associate a formal meaning to their components and interactions.

In cite [6, 1], formal languages are proposed to model component based systems. In both contributions, semantics of languages are more specifically dedicated to model interaction between components than to capture various semantics of component executions. In both case this is done by specifying restrictions of component parallel compositions that underly the language semantics. Our Language and associated symbolic execution mechanisms follow a similar philosophy but also allows one to write model semantics in a more imperative style which may be considered as a more convenient way to write execution rules. The main difference with our proposal is the way we handle time. In cite [6, 1], no specific constructions are proposed to capture time, even though it can be captured by means of components modeling clocks which constrain the system time, even though it can be captured by means of component parallel compositions that underly the language semantics.

Paper organization. In section 2, we recall some notions related to many sorted equational logic, that we use to describe data types. In section 3, we introduce our language and its associated symbolic execution mechanisms. Section 4 is a conclusion.

2 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 to a type by means of a function $type : V \rightarrow S$. The set $T\Omega(V)$ of $\Omega$-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 $\Omega$-substitution over $V$ is a function $\sigma : V \rightarrow T\Omega(V)$ preserving types which can also be canonically extended to $T\Omega(V)$. $T\Omega(V)^{\omega}$ denotes the set of all $\Omega$-substitutions defined on $V$. The set $Sen\Omega(V)$ of all $\Omega$-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. $type(t) = type(t')$.

A many sorted equational specification is a couple $SP = (\Omega, ax)$ where $\Omega$ is a data signature and $ax \subseteq Sen\Omega(V)$ is a finite set.

An $\Omega$-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 $s$. $\Omega$-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 $\varphi$, 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 "$\Omega$" is removed (terms will be used instead of $\Omega$-terms for instance).

3 A symbolic execution based framework for heterogeneous modeling

A design has to be understood as a black box only observable by means of a set of ports which are typed public variables used by the design to exchange data with its environment. One observes design executions by means of successive snapshots which denote observable states of the system. A snapshot structures pieces of information concerning values assigned to ports and possibly a date at which the observation is made (whenever it makes sense in semantics of the corresponding component). Rules which define the succession of snapshots are given by a program declared in the design. A design may structure other designs (we call them reused designs) typically when it defines some interaction semantics (reused designs are those associated to subcomponent semantics). In that case, rules encoded in the program depends on reused design states (i.e. the state of the system depends on the state of its subcomponents). Moreover the program can mod-
ify reused design states by defining communications between reused designs (i.e., state modifications of sub-components may depend on values received on their ports). To reach those purposes, programs introduce specific instructions which allow to observe reused design and define value passing between components. Basic programs are given in the form of Hoare assertions made of: a precondition of application over the snapshots of its associated design and reused designs; a sequence of instructions; a post condition on snapshots. Basic programs may then be scheduled through different operators (sequence/choice/interleaving) allowing to capture a wide class of interaction schemas.

3.1 Design

In the sequel, we suppose that a specification \( SP = (\Omega, ax) \) is given. A design \( D \) is composed of a public part and a private part and a body.

```plaintext
design design_id

public:
  port \( p_1 : [\text{t} = t_1] : s_1, \ldots, p_n : [\text{t} = t_n] : s_n \)
  [time \text{ discrete}|\text{dense}]

private:
  [var \( v_1 : [\text{t} = t_1] : s_1, \ldots, v_m : [\text{t} = t_m] : s_m \)]
  [use design_id_1, \ldots, design_id_k]

body: \( \text{prog} \)
```

design_id is a design identifier. The public part introduces two notions: (1) a set of typed ports which are typed variables; (2) a time carrier which is used to assign dates to observations of port values. That time carrier may be: empty to denote that no date is assigned; discrete ranging over ports, which is isomorphic to natural numbers) or dense (isomorphic to rational numbers)\(^2\). Time is handled as other data: we suppose that the set of types of \( \Omega \) contains \( \text{Time}_{di} \) (discrete) and \( \text{Time}_{de} \) (dense), and the set of operations contains at least the initial date denoted as a constant \( 0_i := \text{Time}_{di} \) for \( i \in \{\text{d}, \text{de}\} \), the function names \( +_i : \text{Time}_{di}, \text{Time}_{di} \rightarrow \text{Time}_{di} \) and \( –_i : \text{Time}_{di}, \text{Time}_{di} \rightarrow \text{bool} \) for \( i \in \{\text{d}, \text{de}\} \). Semantical interpretations of those operations are the usual ones respectively over \( \mathbb{N} \) and \( \mathbb{Q}^+ \). Moreover, we write \( _i \) instead of \( >_i \) for \( i \in \{\text{d}, \text{de}\} \) when the context is clear of confusion, and we also suppose that dates of different carrier may be compared through \( < \) and \( = \) by means of implicit cast operations. In the following we suppose that there exists a unique variable \( \text{time} \) of type \( \text{Time}_{i} \).

Programs introduced in the body section are defined as follows: all public variables \( x \) (ports and time) are prefixed by the identifier of the design declaring them, using the notation \( \text{design_id}_i.x \). Private variables (declared in the field \( \text{var} \)) are not prefixed.

```plaintext

\[
\text{Prog} ::= \text{when}(\text{For}) \{\text{Ins}\} \text{(For)} \ |
\{ p \odot p' \} \text{ with } \odot \in \{\land, \lor, \lnot\} \ |
\text{while}(\text{For})\{\text{Prog}\}
\]

\[
\text{Ins} ::= \text{in}(\text{design_id}_i) \text{ provided}(\text{For}) \ |
\text{d.p} \rightarrow \text{d'.p'} \ |
\text{t} \leftarrow \text{t} \ |
\text{if}(\text{For}) \text{ then} \{\text{Ins}\} \text{ else } \{\text{Ins}\} 
\]

\]

\[
\text{where:}
\]

- \( x \) is a private variable or a port of \( \text{design_id}_i \). \( t \) is a term over \( \Omega \) and the set of variables containing private and public variables of \( D \) and public variables of each design \( D_i \) (\( i = 1, \ldots, k \)) reused by \( D \).

- \( d \in \{\text{design_id}_1, \text{design_id}_2, \ldots, \text{design_id}_k\} \), \( d' \in \{\text{design_id}_1, \ldots, \text{design_id}_k\} \) and \( p \) and \( p' \) are respectively ports of \( d \) and \( d' \).

\[
\text{For} ::= t_1 = t_2 \text{ where } t_1 \text{ and } t_2 \text{ are of same type} \ |
\text{touch}(\text{design_id}_i) \ |
\text{For} \circ \text{For} \text{ with } \circ \in \{\land, \lor\} \ |
\lnot \text{For}
\]

Intuitively basic programs \( \text{when}(\varphi)\{\text{ins}\}(\psi) \) are similar to Hoare assertions built over standard constructions of imperative programming languages such as assignments, conditional instructions or instruction sequences. \( \text{when}(\varphi)(\text{ins})(\psi) \) is a program whose meaning can be understood as follow: when \( \varphi \) is satisfied, the instruction \( \text{ins} \) is executed and after this execution the property \( \psi \) is satisfied. However we have introduced supplementary constructions to deal with interactions by exchanges of data between components:

- \( \text{touch}(\text{design_id}_i.p) \) can be understood as a predicate ranging over ports, which is is evaluated to \( \text{True} \) when a value has been sent from the environment of \( \text{design_id}_i \) to the port \( p \) but not yet assigned to \( \text{design_id}_i.p \) and,

- \( \text{in}(\text{design_id}_i).\text{provided}(\varphi) \) assigns such an available value to \( \text{design_id}_i.p \) when \( \varphi \) is evaluated to \( \text{true} \).

- \( d.p \rightarrow d'.p' \) is in some way the dual instruction. Its purpose is to make available the value assigned to \( p \)

\[^2\]Continuous time does not make sense here because we are dealing with very concrete specifications on which we want to apply symbolic execution techniques.

\[^3\]We also suppose that \( \Omega \) also contains the sort \( \text{bool} \) with constants \( \text{true} \) and \( \text{false} \).
of design $d$ to the reused design $d'$ through the port $p'$.

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

**Example 1** We define a design $dec$ to model a decrypting machine. This decrypting machine is an element of a radio system. $dec$ receives encrypted or decrypted messages through a port in of some type mess. Whenever the message is encrypted, the decrypting machine decrypts it by means of a function $\text{decrypt: mess} \rightarrow \text{mess}$. The decrypted message or the received one if it were not encrypted, is then stored on a port out. The port newout is used to inform the environment of $dec$ that a new message has been treated: newout is assigned by true. Then newout is assigned by false and the design is ready for another cycle. For clarity sake instructions of the form $\text{in}(x)$ provided(true) are replaced by $\text{in}(x)$ and programs of the form $\text{when}(\varphi)\{\text{ins}\}(\text{true})$ are replaced by $\text{when}(\varphi)\{\text{ins}\}$.

```
    design dec
    public:
        port in : mess, out : mess,
        newout := false : bool
    body:
        while(\text{True}) {
            when(\text{touch}(\text{dec.in})){
                in(\text{dec.in});
                if (\text{crypt}(\text{dec.in}) = true)
                    \{ \text{dec.in} := \text{decrypt}(\text{dec.in});
                    \text{dec.out} := \text{dec.in};
                    \text{dec.newout} := \text{true} \}
                \}
            when(\text{true}) { \text{dec.newout} := \text{false} } 
        }
    }
```

The design $dec$ executes as an asynchronous automata which communicates by handshakes without storing massages. It does not introduce any time: executions are purely causal. Now suppose that we want to build a model of a more complex system reusing $dec$. In that system a message is received every 10 milliseconds. Therefore we want to express that the decrypting machine should perform one cycle in less than 10 milliseconds. To reach that purpose we built a new design $env$ that reuses $dec$ and whose purpose is to impose time constraints on $dec$. When a new message is available for the port in of $env$, this message is stored, the current date is stored on a variable $ot$ and the message is sent to $dec$ through its port in. When the value is decrypted by $dec$, the resulting message is stored on port out of $env$. As for $dec$, a port newout is used to inform the environment of $env$ that a new message has been treated. Postcondition of the last when program of the cycle ensures that the whole treatment takes less than 10 milliseconds.

```
    design env
    public:
        port in : mess, out : mess
        newout := false : bool
        time dense
    private:
        var ot : time
    use dec
    body:
        while(\text{True}) {
            when(\text{touch}(\text{env.in})){
                in(\text{env.in}) \text{ot} := \text{env.time}; \text{env.in} \rightarrow \text{dec.in}
                \{ \text{dec.newout} := \text{true} \}
            }
            when(\text{dec.newout} := \text{false}){
                \text{env.out} := \text{dec.out}; \text{env.newout} := \text{true}
            }
            when(\text{true}){
                \text{env.newout} := \text{false}
            }
            ((\text{env.time} - \text{ot}) < 10)
        }
    }
```

### 3.2 Symbolic execution

Symbolic executions are represented as graphs where nodes and transitions are respectively symbolic denotations of system states and state evolutions. Symbolic states are usually defined by the given of symbolic values of variables together with constraints on those symbolic values. Symbolic values are denoted as terms over a supposed given set of fresh variables

$$F = \{ F_x \}_{x \in S_0}$$

disjoint of the set of variables of the program. In our context, states represent observations of the design variables restricted to public ports and dates to which observations are made. Such observations are called snapshots. Public ports are associated with currently assigned values (by means of the function $\eta$). In order to capture communications between designs, ports are also possibly associated with so-called

---

$S_0$ is the set of types introduced in $\Omega$. 

4
available values (by means of the function \( \nu \)) which represent pieces of data sent from the design environment through public ports. We note \( P(D) \) the set of port variables \( \text{design}_id.p \) of a given design \( D \).

**Definition 1 (Snapshots)** A snapshot over \( D \) is any element \((\eta, \nu, \delta, \pi)\) of \((T_\Omega(F))^{P(D)} \times (T_\Omega(F) \cup \{\bot\})^{P(D)} \times (T_\Omega(F)_{time_i} \cup \{\varepsilon\}) \times 2^{\text{Sen}(F)}\) satisfying:

\[
\delta \in T_\Omega(F)_{time_i}, \text{ if } D \text{ declares } time_i,
\]

\[
\delta = \varepsilon \text{ otherwise,}
\]

and \( \pi \) is finite. We note \( \text{Snap}(D) \) the set of all snapshots over \( D \).

\( \bot \) is used to model the absence of available value. \( \pi \) is a set of constraints computed along the execution leading to the snapshot. In the sequel for any such snapshot \( s \), the notations \( \eta_s, \nu_s, \delta_s \) and \( \pi_s \) stand respectively for \( \eta, \nu, \delta \) and \( \pi \).

The symbolic execution of a design \( D \) is a particular structure, so-called execution graph of \( D \).

**Definition 2 (Execution graph)** An execution graph of \( D \) is a couple \((\text{Init}, R)\) where \( \text{Init} \in \text{Snap}(D) \) and \( R \subseteq \text{Snap}(D) \times \text{Snap}(D) \) such that the following conditions hold:

(Initialization). \( \eta_{\text{Init}} \) is injective, \( \pi_{\text{Init}} = \emptyset \), and 
\[
\forall x \in P(D), \eta_{\text{Init}}(x) \in \mathcal{F} \wedge \nu_{\text{Init}}(x) = \bot.
\]

(Time). \( \forall (s_1, s_2) \in R, \)
\[
(\delta_{s_1} = \delta_{s_2} = \varepsilon) \vee (\delta_{s_2} > \delta_{s_1} \in \pi_{s_2})
\]

(Consumption). \( \forall (s, s') \in R, \forall x \in P(D), \)
\[
\nu_s(x) \neq \bot \Rightarrow \nu_{s'}(x) \in \{\nu_s(x), \bot\}
\]

One defines \( G(D) \) as the set of all execution graphs of \( D \).

Consumption condition ensures that consumptions of available values are observable in execution graphs.

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 an associated snapshot together with observations of its reused design states (i.e. snapshots of reused designs). Such a complete state observation is called a synchronization. In the sequel, \( V(D) \) refers to the set of private variables declared in \( D \).

**Definition 3 (Synchronization)** Let \( D_1, \ldots, D_k \) be the reused designs of \( D \). A \( D \)-synchronization is a triple \( \text{sync} = (s, i, \gamma) \) where \( s \in \text{Snap}(D), i : V(D) \rightarrow T_\Omega(F) \) is a mapping, and \( \gamma \in \prod_{j \leq k} \text{Snap}(D_j) \).

We note \( \gamma_j \) for the \( j \)-th-projection of \( \gamma \) and \( \text{Sync}(D) \) the set of all \( D \)-synchronizations.

Note that besides 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 a given synchronization of \( D \). This substitution associates a symbolic value to any public or private variable of \( D \) and only to public variables of reused designs. If we note for any design \( D' \), \( \text{Time}(D') \) the singleton containing the variable \text{design}_id.time if \( D' \) declares a time and the empty set otherwise, then we define \( \text{Var}(D) = P(D) \cup V(D) \cup \text{Time}(D) \cup \bigcup_{D' \subset R(D)} (p(D') \cup \text{Time}(D')) \).

**Definition 4 (Substitution)** With the notations of Definition 3, the substitution associated to \( \text{sync} \), denoted \( \iota_{\text{sync}} : \text{Var}(D) \rightarrow T_\Omega(F) \) is defined as follows:

- \( \forall x \in P(D), \iota_{\text{sync}}(x) = \eta_s(x) \).
- \( \forall x \in V(D), \iota_{\text{sync}}(x) = i(x) \).
- \( \text{Time}(D) \neq \emptyset \Rightarrow \iota_{\text{sync}}(\text{design}_id.time) = \delta_s \).
- \( \forall j \leq k, \big\{ \forall x \in P(D_j), \iota_{\text{sync}}(x) = \eta_j(x) \big\} \big\{ \forall x \in \text{Time}(D_j), \iota_{\text{sync}}(x) = \delta_j \big\} \).

We also note \( \iota_{\text{sync}} \) the canonical extension to terms and formulas which associates to any formula \( \varphi \) the formula \( \iota_{\text{sync}}(\varphi) \) obtained by replacing: (1) occurrences of any ports or variables \( x \) by \( \iota_{\text{sync}}(x) \); (2) sub-formulas of the form \( \text{touch}(\text{design}_id.x) \) (resp. \( \text{touch}(\text{design}_id_{i,j}x) \)) with \( j \leq k \) by \( \text{True} \) if \( \nu_s(\text{design}_id.x) \neq \bot \) (resp. \( \nu_{i,j}(\text{design}_id_{i,j}x) \) \( \neq \bot \)) and \( \text{False} \) otherwise. Let us notice that \( \text{touch}(x) \) 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.

**Definition 5 (Instruction symbolic execution)** For \( j \leq k \), let us note \( G_j \) an execution graph of the design \( D_j \). Let us note \( G = (G_j)_{j \leq k} \). For any instruction \( g \), let us note \( [g]_G \subseteq \text{Sync}(D) \times \text{Sync}(D) \) the symbolic execution of \( g \) inductively defined on the structure of instructions \((s, i, \gamma) [g]_G (s', i', \gamma') \) if, and only if \( \delta_{s'} = \delta_s \) and:

- if \( g \equiv x := t \) then \( \gamma' = \gamma \).
- if \( x \in P(D) \), then\(^5\) \( \eta_{s'} = \eta_{s,x \rightarrow a} \) for some fresh variable \( a \in F \), \( \nu_{s'} = \nu_s \), \( \iota' = \iota \) and \( \pi_{s'} = \pi_s \cup \{ a = \iota_{(s,i,\gamma)}(t) \} \)
- if \( x \in V(D) \) then \( \iota' = i_{x \rightarrow a} \) 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 \( \varphi \equiv \text{in}(x)\text{provided}(\varphi) \), then \( \nu_s(x) \neq \bot \), \( \gamma' = \gamma \).

\( \eta_{s'} = \eta_{x \rightarrow a} \) for some \( a \in F \), \( \nu_{s'} = \nu_{x \rightarrow \bot} \) and\(^6\) \( \pi_{s'} = \pi_s \cup \{ \{ t_{(s,i,\gamma)}(\varphi) \}_{x \rightarrow a} \land a = \nu_s(x) \} \).

- if \( \varphi \equiv x \rightarrow \text{design}_{-} \text{id}_{-} \cdot \text{p'} \), then:
  - \( \nu_{\gamma_j}(\text{design}_{-} \text{id}_{-} \cdot \text{p'}) = \bot \),
  - for all \( i \neq j \leq k \gamma_i = \gamma_t \),
  - if we note \( G_j = (I_j, R_j) \) then \( \gamma_j R_j \gamma'_{j} \),
  - \( \nu_{\gamma_j} = \eta_{\gamma_j} \), \( \nu_{\gamma'_j} = \nu_{\gamma_j} \text{design}_{-} \text{id}_{-} \cdot \text{p'} \rightarrow \iota_{x \rightarrow a} \) and
  - \( s' \) is as \( s \) except that \( \pi_{s'} = \pi_s \cup \{ a = \iota_{(s,i,\gamma)}(x) \} \).

- sequences and conditions are handled as usual.

\text{in}(x)\text{provided}(\varphi) \) cannot be executed from a synchronization where no value is made available for \( x \) (this reflects to a deadlock situation) and otherwise results on a synchronization which reflects the assignment of \( x \) by the available value. Snapshots of reused designs are not modified. \( x \rightarrow \text{design}_{-} \text{id}_{-} \cdot \text{p'} \) cannot be executed from a synchronization where a value is already available for \( x \) (this also reflects a deadlock situation). Otherwise, it yields a set of synchronizations \( (s', i, \gamma') \) where all snapshots of \( \gamma' \) but the one associated to the design \( D_j \) are the ones declared in \( \gamma \) (states of the corresponding reused designs are not modified by the instruction execution).

The snapshot corresponding to \( D_j \) is chosen in the set of snapshot reachable from \( \gamma_j \) in the execution graph \( G_j \) of \( D_j \). The chosen snapshot reflects that a new value (symbolically denoted by \( a \)) is available for \( x \): if none exists, the instruction execution results on a deadlock. Finally, \( s' \) is \( s \) except that the constraint \( a = \nu_{\text{sync}(x)} \) is added to reflect that the available value is the one assigned to \( x \) in \( \text{sync} \).

Definition 5 only defines how the execution of instructions modifies the state of a design. Possible resulting states are denoted by the set of resulting synchronizations. However, from such a resulting synchronization, the snapshot of the design as well as those corresponding to reused designs may be modified without any instruction execution. The main design state (i.e. snapshot) may evolve because both time passes and new values may become available for some of its ports. To capture this fact we define the notion of stuttering of a snapshot.

**Definition 6 (Stuttering)** The stuttering of a snapshot \( s \), denoted \( \text{St}(s) \), is the set of snapshots \( s' \) such that \( \eta_{s'} = \eta_s \) and:

- for all \( x \) s.t. \( \nu_s(x) \neq \bot \) then \( \nu_{s'}(x) = \nu_s(x) \)
- there exists at most one \( x \) such that \( \nu_s(x) = \bot \) and \( \nu_{s'}(x) \neq \bot \): for this \( x \), \( \nu_{s'}(x) \in F \).

Moreover, if:

\( \delta_s = \varepsilon \) then \( \delta_{s'} = \varepsilon \) and \( \pi_{s'} = \pi_s \)

\( \delta_s = \neq \varepsilon \) then \( \delta_{s'} = a \) and \( \pi_{s'} = \pi_s \cup \{ a > \delta_s \} \) with \( a \in F \)

Snapshots of reused designs may also evolve due to some internal executions. Those internal executions correspond to state evolutions that do not require the availability of new value for ports of the reused design: making such new values available is the result of an operation which is external from the reused design point of view. The set of resulting snapshot is defined as follows.

**Definition 7 (Internally reachable snapshots)** Let \( s \) be a snapshot of some execution graph \( G = (\text{Init}, R) \). The set of snapshots internally reachable from \( s \) in \( G \), denoted \( T_s \subseteq \text{Sn}(D) \times \text{Sn}(D) \) is the transitive closure of:

\[ (s, R, s') \iff \forall x \in P(D), \nu_s(x) \neq \bot \Rightarrow \nu_{s'}(x) = \nu_s(x) \]

Intuitively \( s' \) 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 \( \nu_s(x) \neq \bot \) and \( \nu_{s'}(x) = \bot \) this reflects a consumption of the value available for \( x \). Property (Consumption) of definition 2 ensures that all such consumptions are observable.

Mixing state evolutions denoted by Definitions 6 and 7, we define the set of so called uncontrollable futures of a synchronization.

**Definition 8 (Uncontrollable futures)** Let \( \text{sync} = (s, i, \gamma) \) be a \( D \)-synchronization and for every \( j < k \) let \( G_j \) be an execution graph of \( D_j \). The Uncontrollable futures of \( \text{sync} \), denoted \( \text{F}^*(\text{sync}) \) is the set of all \( D \)-synchronizations \( \text{sync}' = (s', i, \gamma') \) s.t. \( s' \in \text{St}(s) \) and for all \( j < k \), \( \gamma_j \subseteq \gamma_j' \).
We now define the program symbolic execution. The program symbolic executions relies on the symbolic executions of instructions introduced in the program. Given a set of execution graphs $\mathcal{G} = (\mathcal{G}_i)_{i \leq k}$, the symbolic execution of instructions is a binary relation on synchronizations. Therefore, the symbolic execution of programs is of the same form.

We introduce the following technical notations: in the sequel, for any set $E$ and relation $R \subseteq E \times E$ we note $L[R]$ the set of leaves of $R$ defined as $\{(y \exists x \in E. x R y) \land \forall z \in E. \neg(y R z)\}$.

Definition 9 (Program symbolic execution) With notations of Definition 5, for every program $\alpha$, the symbolic execution of $\alpha$ is the relation $[\alpha]_\mathcal{G} \subseteq \text{Sync}(\mathcal{D}) \times \text{Sync}(\mathcal{D})$, such that:

- $[\text{when}(\psi)\{\rho\}(\psi)]_\mathcal{G}$ is the set of all $(\text{sync}, \text{sync}')$ such that if we note $\text{sync}_1$ the synchronization $(s_1, i_1, \gamma_1)$ defined as $\text{sync} = (s, i, \gamma)$ except that $\pi_{s_1} = \pi_s \cup \{t_{\text{sync}}(\psi)\}$, then
  - there exists a synchronization $\text{sync}_2$ such that $\text{sync}_1 \cap \text{sync}_2 \neq \emptyset$ and
  - if we note $\text{sync}_3$ the synchronization defined as $\text{sync}_2$ except that $\pi_{s_2} = \pi_s \cup \{t_{\text{sync}}(\psi)\}$, then $\text{sync} \cap \text{sync}_3 \neq \emptyset$.

- $[\text{while}(\psi)\{\alpha\}]_\mathcal{G} = \bigcup \{\{\text{sync}, \text{sync}_-\psi\} \cup \{\text{sync}, \text{sync}_\psi\}\}$ where the notation $\text{sync}_\psi$ for $\psi$ a formula, is the synchronization $(s', i, \gamma)$ obtained from $\text{sync} = (s, i, \gamma)$ by replacing $\pi_s$ by $\pi_{s'} = \pi_s \cup \{t_{\text{sync}}(\psi)\}$.

Notice that in Definition 9, for a given program $\alpha$ and a given input synchronization, we are not only interested in final synchronizations obtained after having performed $\alpha$: we are also interested in all the intermediate synchronizations resulting from the performance of all sub-programs of $\alpha$. They are needed to define symbolic executions of designs.

Definition 10 (Symbolic execution of designs) Let $\mathcal{D}$ be a design, $\mathcal{D}_1, \ldots, \mathcal{D}_k$ be its reused designs, and $\alpha$ 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 $(s, i, \gamma)$ be a 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 $i_1, i_2 : V(\mathcal{D}) \to T_{\mathcal{D}}(F)$ and $\gamma_1, \gamma_2 \in \bigcup_{j \leq k} \text{Snp}(\mathcal{D}_j)$ such that $(s_1, i_1, \gamma_1) [\alpha]_\mathcal{G} (s_2, i_2, \gamma_2)$.

Example 2 Figure 1 denotes a path of the symbolic execution of env (Example 1). Since dec is reused in env, this symbolic execution takes the symbolic execution of dec as the execution graph associated to dec. The path depicted in Figure 1 corresponds to a scenario in which:

- an encrypted message env.in$_1$ is received on the port env.in,
- env.in$_1$ is then sent to dec which decrypts it,
- the resulting decrypted message env.out$_1$ is stored on env.out and thus is visible for the environment of env.

The set of constraints appearing in the last snapshot env$_4$ contains:

- env.out$_1 = \text{decrypt}(a)$ which ensures that env.out$_1$ is the decrypted message. $a$ is the variable used in the symbolic execution of dec to denote incoming messages. The formula $\alpha = \text{env.in}_1$ occurs in $\pi_{\text{env}_s}$, which is note given in extension here. This formula ensures that a corresponds to env.in$_1$.
- $t_4 - t_1 < 10$ where $t_4$ is the date at which the new message is received by env and $t_4$ is the date at which the decrypted message was available on env.out.

For this scenario, we observe the effect of the constraints imposed by env on dec in Example 1.
Figure. 1

Theorem 1 For any design $\mathcal{D}$ $G(\mathcal{D})$ is an execution graph.

Theorem 1 holds because (Initialization condition) is ensured by Definition 10, (Time) is ensured by the Definitions 9 and 8. Finally (Consumption) is ensured by the definition of symbolic executions of intructions of the form $in(x)/provided(\varphi)$ in Definition 5. Theorem 1 is very important since it ensures the ability to recursively symbolically execute designs built in a hierarchical manner. In other words $\mathcal{D}$ can be reused in some other designs whose symbolic executions take $G(\mathcal{D})$ as execution graph associated to $\mathcal{D}$.

Finally, we show how to associate semantics to any design $\mathcal{D}$ by means of its associated symbolic execution. Such semantics are defined as sets $\overline{\mathcal{D}}$, whose elements are called finite execution paths of $\mathcal{D}$. Finite execution paths are sequences of so-called numerical snapshots. Numerical snapshots are defined up to a model $M \in Mod(SP)$ and are triples $s = (\eta_s, \nu_s, \delta_s) \in oFM^{(\mathcal{D})} \times (M \cup \{\bot\})^{\mathcal{D}} \times (M_{time} \cup \{\varepsilon\})$. A finite execution path $p = s_1 \cdots s_n$ corresponds intuitively to a sequence of observations of observable states of $\mathcal{D}$ 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 $s' p$ of $p$ if and only if $\nu_x(p) = \bot$ and $\nu_x(p) = v$. With notations of Definition 10, $\overline{\mathcal{D}}$ is the set of all finite execution paths $s_1 \cdots s_n$ of $\mathcal{D}$ such that 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:

- $\exists i : F \rightarrow M$ such that $M \models i \land_{\varphi \in \pi_s} \varphi$ 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 $\nu_{s_k}(x) \neq \bot$ and $\nu_{s_k}(x) = \bot$ otherwise, and finally $\delta_{s_n} = i(\delta_{s'_n})$.

### 4 Conclusion

We have proposed a framework to design and symbolically execute heterogeneous models of systems. The framework consists in a language $LEM$ allowing both to take into account semantics of basic component models and to express interaction between them. Time and time synchronizations are taken into account as first-order constraints. $LEM$ introduces basic instructions allowing to denote exchanges of values between components. Those instructions structured in programs are used to describe different models of interaction. Basic programs are given in the form of Hoare assertions that can be scheduled by means of different operators (sequence/choice/interleaving) allowing to capture a wide class of interaction schemas. We have shown how to apply symbolic execution to heterogeneous models. This is done by defining symbolic execution mechanism dedicated to $LEM$. $LEM$ has been implemented as a meta-model extension of $UML$ in the Papyrus tool set developed at CEA LIST. The goal is to use $LEM$ to express formally various semantics of execution of various $UML$-views and to define models of interaction between them. The symbolic execution mechanisms are being currently implemented in the tool set $AGATHA$ [12] developed at CEA LIST in the frame of the national project $ANR Hecosim$ dedicated to heterogeneous system testing and simulation. We currently work on extending the symbolic execution based testing framework proposed in [7, 4, 5] to black box testing of heterogeneous systems.

### References


