Proceedings of the 17th World Congress
The International Federation of Automatic Control
Seoul, Korea, July 6-11, 2008

Concrete syntax and semantics of the compositional interchange format for hybrid systems ⋆

D.A. van Beek * **M.A. Reniers** † **J.E. Rooda** * **R.R.H. Schiffelers** *

* *Department of Mechanical Engineering*
† *Department of Mathematics and Computer Science*
Eindhoven University of Technology, P.O.Box 513, 5600 MB Eindhoven, The Netherlands
{*d.a.v.beek, m.a.reniers, j.e.rooda, r.r.h.schiffelers*}*@tue.nl*

Abstract: The compositional interchange format for hybrid systems is syntactically and semantically defined in terms of an interchange automaton in an abstract format, allowing among others differential algebraic equations, variables that can be internal or external, operators for parallel composition, action hiding, variable hiding and urgent actions, synchronization by means of shared labels, and communication by means of shared variables and CSP channels. A concrete format is defined for modeling. Its semantics is defined in terms of a mapping to the abstract format. The concrete format adds inputs, outputs and open and closed scopes to enable modular and hierarchical specifications. The concrete format is illustrated by means of a bottle filling line example.

1. INTRODUCTION

In Beek et al. [2007a] and Beek et al. [2007b] the foundations of a compositional interchange format for hybrid systems (CIF) have been defined, along with a detailed discussion of design considerations, and the CIF has been related to previous work on interchange formats for hybrid systems: MoBIES team [2002], Pinto et al. [2006], Cairano et al. [2006]. The main requirements for the interchange format, as defined in Beek et al. [2007a], are summarized below.

(1) It should have a formal and compositional semantics, based on (hybrid) transition systems, and allow property preserving model transformations.
(2) Its concepts should be based on mathematics, and independent of implementation aspects such as equation sorting, and numerical equation solving algorithms.
(3) It should support arbitrary differential algebraic equations (DAEs), including fully implicit equations, higher index systems, algebraic loops, steady state initialization, switched systems such as piecewise affine systems, and DAEs with discontinuous right hand sides.
(4) It should support a wide range of urgency concepts, such as used in hybrid automata, including ‘urgency predicates’, ‘deadline predicates’, ‘triggering guard semantics’, and ‘urgent actions’.
(5) It should support parallel composition with synchronization by means of shared variables and shared actions.
(6) It should support hierarchy and modularity to allow the definition of parallel modules and modules that can contain other modules (hierarchy), and to allow the definition of variables and actions as being local to a module, or shared between modules.

⋆ Work partially done in the framework of the HYCON Network of Excellence, contract number FP6-IST-511368; as part of the Darwin project under the responsibility of the Embedded Systems Institute, partially supported by the Netherlands Ministry of Economic Affairs under the BSIK program; and as part of the ITEA project Twins 05004.

The contribution of this article is twofold:

(1) The abstract syntax of the CIF, as defined in Beek et al. [2007a] is extended with *channels* to allow a CSP style communication (see Hoare [1985]), such as used in the Chi language (see Beek et al. [2006], Man and Schiffelers [2006]) and as used in UPPAAL (see Larsen et al. [1997]), although the latter restricts CSP to pure synchronization (no communication of data).

(2) A *concrete format*, that is used for modeling, is defined. The semantics of the concrete format is formally defined by means of a mapping to the abstract format (as defined in Beek et al. [2007a]). The language elements of the abstract format are mathematical constructs, such as sets, partial functions etc. chosen to facilitate the definition of the semantics. It consists of a small number of orthogonal language elements. For modelling purposes, however, the abstract syntax is rather cumbersome. The concrete syntax is chosen to facilitate modelling. For example, instead of defining a set of variables, a partial function to define their dynamic type (see Section 2), and a predicate defining their initial values, a variable declaration mechanism as used in many modelling languages is used. Furthermore, the concrete syntax extends the abstract syntax with constructs for modeling, including amongst others

- *clocks* that are added for compatibility with timed automata,
- *input and output variables* that are added for compatibility with languages such as Simulink (see The MathWorks, Inc [2005]) and PHAVER (see Frehse [2005]), and to enable compositional verification in the form of assume-guarantee reasoning (e.g. see Henzinger et al. [2000], Frehse et al. [2001]),
- *open and closed scopes* that allow the definition of variables, channels, clocks and actions as being local to facilitate hierarchy and modularity,
- *automaton definition and instantiation* that facilitate reuse of automata.

The remainder of this article is organized as follows: Section 2 defines the abstract syntax of the CIF, Section 3 informally explains the semantics of the abstract syntax, Sections 4 and 6 define the concrete syntax and its mapping to the abstract syntax, respectively, Section 5 presents a bottle filling system example and Section 7 presents concluding remarks.

2. ABSTRACT SYNTAX OF INTERCHANGE AUTOMATA

First some notations are defined. A set $\mathcal{V}$ of variables, a set of basic action labels $\mathcal{L}_{\text{basic}}$, which does not include the predefined non-synchronizing action τ, a set of channel labels $\mathcal{H}$, and a set of values Λ are assumed. The set $\mathcal{L}_{\text{com}}$ denotes the set of CSP action labels. It is defined as $\mathcal{L}_{\text{com}} = \{h!cs, h?cs, h!?cs \mid h \in \mathcal{H}, cs \in \Lambda^*\}$, where $h \in \mathcal{H}$ denotes a channel, and $cs \in \Lambda^*$ denotes a list $[c_1, \ldots, c_n]$ of values ($c_i \in \Lambda$, $1 \leq i \leq n$). The CSP actions labels $h!cs$, $h?cs$, $h!?cs$ are called send action label, receive action label, and communication action label, respectively. We assume the set of basic action labels and the set of CSP action labels to be disjoint: $\mathcal{L}_{\text{basic}} \cap \mathcal{L}_{\text{com}} = \emptyset$. The set $\mathcal{L}$ denotes the set of basic and CSP action labels $\mathcal{L}_{\text{basic}} \cup \mathcal{L}_{\text{com}}$, and the set $\mathcal{L}_\tau$ denotes the set $\mathcal{L} \cup \{\tau\}$. For a set of variables $S \subseteq \mathcal{V}$, Pred(S) denotes the set of all predicates over variables from S, and Expr(S) denotes the set of all expressions over variables from S.

Definition 1. (Atomic Interchange Automaton). An *atomic interchange automaton* is a tuple (X, X_{i}, dtype, V, v_0, init, flow, inv, tcp, L, E) where

- $X \subseteq \mathcal{V}$ is a finite set of variables, $X_{\text{i}} \subseteq X$ is the set of *internal* variables.
- dtype : $X \to$ {disc, cont, alg} is a function that associates to each variable a dynamic type: *discrete*, *continuous* or *algebraic*. The sets X_{disc}, X_{cont}, X_{alg} are defined as $X_t = \{x \in X \mid \mathsf{dtype}(x) = t\}$ for $t \in$ {disc, cont, alg}, and $X_{\text{state}} = X_{\text{disc}} \cup X_{\text{cont}}$ is the set of *state* variables.
- V is a finite non-empty set of *vertices*, called *locations*, and $v_0 \in V$ is the initial location.
- init $\in$ Pred($\tilde{X}$) is the initial condition. For $Y \subseteq X$, $\tilde{Y} = Y \cup \{\dot{y} \mid y \in Y \cap X_{\text{cont}}\}$ is the extension of Y with the dotted versions of the continuous variables in Y.
- flow, inv, tcp : $V \to$ Pred($\tilde{X}$), are functions that each associate to each location $v \in V$ a predicate describing the *flow condition*, the *invariant*, and the *time can progress condition*, respectively.
- $L \subseteq \mathcal{L}_{\text{basic}}$ is a finite set of synchronizing action labels. Usually, this set includes at least the labels that occur on the edges of the automaton, in which case the set L is referred to as the *alphabet* of the automaton.
- $E = V \times \text{Pred}(\tilde{X}) \times (\mathcal{L}_{\text{basic}} \cup \{\tau\} \cup C_X) \times (\mathcal{P}(\tilde{X}) \times \text{Pred}(\tilde{X} \cup \tilde{X}^-)) \times V$ is a finite set of *edges*, such that for each element $(v, g, a, (W, r), v') \in E$, v and v' are the *source* and *target* locations, respectively, g is the *guard*, a is an *action statement*, $W \subseteq \tilde{X}$ is a set of jumping variables (the value of which may change as a result of an action transition), and r is the *jump predicate*, also called *reset map*. For any $Y \subseteq \mathcal{V} \cup \dot{\mathcal{V}}$, $Y^- = \{y^- \mid y \in Y\}$ denotes the set of minus superscripted variables that represent the values of variables before an action transition. Three kinds of action statements exist: basic action labels $a \in L$ that synchronize on the basis of equality, the predefined non-synchronizing τ action, and CSP statements $a \in C_X$, where $C_X = \{\ h!\mathbf{e},\ h?\mathbf{x},\ h!?,\ h!?\mathbf{x} := \mathbf{e} \mid h \in \mathcal{H},\ \{\mathbf{e}\} \subseteq \text{Expr}(\tilde{X}),\ \{\mathbf{x}\} \subseteq \tilde{X}\ \}$, where $\mathbf{e}$ and $\mathbf{x}$ are either empty ($n = 0$) or denote comma separated sequences $e_1, \ldots, e_n$ and $x_1, \ldots, x_n$ of expressions and variables ($n \geq 1$), respectively. The CSP statements $h!\mathbf{e}$, $h?\mathbf{x}$, $h!?$, $h!?\mathbf{x} := \mathbf{e}$ are called send statement, receive statement, synchronization statement, and communication statement, respectively. We assume the set of CSP statements to be disjoint from the set of basic action labels: $C_X \cap \mathcal{L}_{\text{basic}} = \emptyset$.

The interchange automaton format consists of automata, and operators for parallel composition, for hiding of actions and variables, and for the definition of urgent actions. The automata and operators can be freely combined:

Definition 2. (Interchange automaton). The set of interchange automata $\mathcal{A}$ is defined by the following grammar for the interchange automata $\alpha \in \mathcal{A}$:

$\alpha ::=$	α_{atom}	atomic interchange automaton
$\mid$	$\alpha \parallel \alpha$	parallel composition
$\mid$	$\mathsf{hidevar}_{X_{\text{h}}}(\alpha, \sigma_{\text{h}})$	variable hiding operator
$\mid$	$\mathsf{hideact}_{L_{\text{h}}}(\alpha)$	action hiding operator
$\mid$	$\mathsf{urgent}_{L_{\text{u}}}(\alpha)$	urgent action operator
$\mid$	$\mathsf{encap}_{L_{\text{e}}}(\alpha)$	action encapsulation operator,

where

- α_{atom} denotes an atomic interchange automaton;
- $X_{\text{h}} \subseteq \mathcal{V}$ denotes a set of variables to hide and $\sigma_{\text{h}} : X_{\text{h}} \mapsto \Lambda$ denotes a (partial) valuation for the hidden state variables of interchange automaton α;
- $L_{\text{h}} \subseteq \mathcal{L}$ denotes a set of actions to hide;
- $L_{\text{u}} \subseteq \mathcal{L}_\tau$ denotes a set of nondelayable actions;
- $L_{\text{e}} \subseteq \mathcal{L}$ denotes a set of actions that are blocked.

3. SEMANTICS OF THE ABSTRACT SYNTAX

The informal semantics of the abstract syntax is defined below. The complete formal semantics, including CSP channels, is defined in Beek et al. [2007b]. The semantics without CSP communication has appeared in Beek et al. [2007a].

3.1 Atomic automata

Variables The interchange automaton defines three classes of variables: the discrete and continuous variables, and in addition the algebraic variables. The main differences are as follows: First, the values of discrete variables remain constant when model time progresses, the values of continuous variables may change according to a continuous function of time when model time progresses, and the values of algebraic variables may change according to a discontinuous function of time. Second, the values of the discrete and continuous variables do not change in action transitions unless such changes are explicitly specified, for example by assigning a new value. The values of algebraic variables can change arbitrarily in action transitions, unless such changes are explicitly restricted, for example by assigning a new value. Third, there is a difference between the different classes of variables with respect to how the resulting values of the variables in a transition relate to the starting values of the variables in the next transition. The resulting value of a discrete or continuous variable in a transition always equals its starting value in the next transition. For algebraic variables there is no such relation. In most models, the values of discrete variables are defined by assignments, whereas the values of algebraic variables are defined by invariants ((in)equalities).

Predicates in a location The initial condition should hold initially, whereas the invariant should hold at all times. The time can progress (tcp) predicate allows passing of time in a location for as long as the condition is true, or in other words, until the time-point when the condition is false. Differential algebraic equations (DAEs) can be specified in the invariants of an interchange automaton, since such invariants are predicates over all variables, including the dotted variables. Flow clauses are supported for reasons of compatibility with existing hybrid automata. The reason for not enforcing a separation between invariants (over non-dotted variables) and flow clauses (over dotted variables), as in existing hybrid automata, is that such a separation is absent in the mathematical theory of dynamical systems, including control theory. In many cases, fully implicit DAEs, cannot even be rewritten to a form where the algebraic constraints and the differential constraints are separated.

In the formal semantics, three kinds of transitions are defined for interchange automata: action transitions, time transitions and consistency transitions. Action transitions and time transitions are well known in hybrid automata. Consistency transitions in the CIF ensure that when one of the automata does an action transition in a parallel composition, the initial conditions and invariants of the automata that do not synchronize hold.

3.2 Operators

Parallel composition operator There are no compatibility requirements for the parallel composition of interchange automata: any pair of interchange automata can be composed by the parallel composition operator. The parallel composition operator synchronizes on external actions that the arguments share. CSP actions synchronize on the basis of pairs of send ($h!e$) and receive ($h?x$) actions. The CSP communication action $h!?x := e$ is the result of elimination of parallel composition as defined in Beek et al. [2007b]. All other actions may be interleaved (under the condition that they maintain the consistency of the other automaton). Time transitions must be synchronized, and consistency is established only if both automata agree on it. The external state variables that are shared by the argument automata need to have the same values (all the time).

Hiding operators The action hiding operator applied to an automaton, $\mathsf{hideact}_{L_\mathrm{h}}(\alpha)$, hides (abstracts from) the actions from set L_h by replacing them by the internal action τ. This only affects the action behavior of α; its delay behavior and consistency remain unchanged. Transitions contain, amongst others, information about the variables, such as their values or, in case of a time transition, their trajectories. The variable hiding operator applied to an automaton, $\mathsf{hidevar}_{X_\mathrm{h}}(\alpha, \sigma_\mathrm{h})$, hides the variables from set X_h by removing the information about them from the transitions of α. The values of the hidden state variables after a transition are stored in valuation σ_h.

Urgent action operator The urgent action operator applied to an automaton, $\mathsf{urgent}_{L_\mathrm{u}}(\alpha)$, gives actions from the set L_u priority over time passing. The action behavior and consistency of α are not affected by the urgent action operator. Time transitions are allowed only if at the current state, and at each intermediate state while delaying, no actions from the set L_u are possible.

Action encapsulation operator The action encapsulation operator applied to an automaton, $\mathsf{encap}_{L_\mathrm{e}}(\alpha)$, blocks actions from the set L_e. The delay behavior and consistency behavior of α are not affected. Send and receive actions on channels from a set H_e can be blocked by means of $\mathsf{encap}_{\{h!cs, h?cs \mid h \in H_\mathrm{e}, cs \in \Lambda^*\}}(\alpha)$. In this way, only the the synchronous execution of matching send and receive actions via channels from the set H_e can take place.

4. CONCRETE SYNTAX DEFINITION

In this section, the concrete syntax of CIF models is defined using a Backes-Naur (BNF) like notation. The symbol | defines choice, and notation $[Z]$ defines Z as being optional.

spec	::=	[*autDefs*] *model* [*autDefs*]
autDefs	::=	*autDef* | *autDefs autDef*
autDef	::=	automaton autId ['('*paramDecls* ')'] = *closedScope*
paramDecls	::=	*paramDecl* | *paramDecls*, *paramDecl*
paramDecl	::=	*varIds* : type
varIds	::=	varId | *varIds*, varId
closedScope	::=	⟦ [*cScopeDecls* ::] *automaton* ⟧
cScopeDecls	::=	*cScopeDecl* | *cScopeDecls*, *cScopeDecl*
cScopeDecl	::=	extern *decls*
		| input var *inputVarDecls*
		| output var *varDecls*
		| intern *decls*
		| connect *connectSets*
decls	::=	*decl* | *decls*, *decl*
decl	::=	var *varDecls*
		| clock *clockIds*
		| chan *chanDecls*
		| act *actIds*
varDecls	::=	*varDecl* | *varDecls*, *varDecl*
varDecl	::=	*varIds* : (disc | cont | alg) type [= (expr | '(' *exprs* ')')]
exprs	::=	expr | *exprs*, expr
clockIds	::=	clockId | *clockIds*, clockId
chanDecls	::=	*chanDecl* | *chanDecls*, *chanDecl*
chanDecl	::=	*chanIds* [! | ?] : type
chanIds	::=	chanId | *chanIds*, chanId
actIds	::=	actId | *actIds*, actId
inputVarDecls	::=	*varDecl* | *varDecls*, *varDecl*
inputVarDecl	::=	*varIds* : type
connectSets	::=	{*connectors*} | *connectSets*, {*connectors*}
connectors	::=	*connector* | *connectors*, *connector*
connector	::=	[autId.] (varId | clockId | chanId | actId)
automaton	::=	*cAutomaton*
		| *oAutomaton*
cAutomaton	::=	[autId:] *closedScope*
		| [autId:] *autInst*
		| *cAutomaton* ‖ *cAutomaton*
autInst	::=	autId ['(' *exprs* ')']
oAutomaton	::=	*atomicAut*
		| [autId:] *openScope*
		| *oAutomaton* ‖ *oAutomaton*
atomicAut	::=	⦇ [*init*,] mode *modes* :: modeId ⦈
init	::=	init *preds*
preds	::=	pred | *preds* & pred
modes	::=	*mode* | *modes*, *mode*
mode	::=	modeId = [*dyns*] [*edges*]
dyns	::=	*dyn* | *dyns dyn*

dyn	::=	(inv | flow | tcp) *preds*
edges	::=	*edge* | *edges edge*
edge	::=	[*guard*] [now] [*action*] [*update*] goto modeId
guard	::=	when *preds*
action	::=	act (actId | *comLabel*)
comLabel	::=	chanId ! [*exprs*]
	|	chanId ? [*varIds*]
	|	chanId !? [*varIds* := *exprs*]
update	::=	do *varClockIds* (:= *exprs* | : *preds*)
varClockIds	::=	*varClockId*
	|	*varClockIds*, *varClockId*
varClockId	::=	varId
	|	clockId
openScope	::=	|([*oScopeDecls* ::] *oAutomaton*)|
oScopeDecls	::=	*oScopeDecl* | *oScopeDecls*, *oScopeDecl*
oScopeDecl	::=	intern *decls*
model	::=	model modelId = *closedScope*

Here, modelId denotes a model identifier, autId denotes an automaton identifier, varId denotes a variable identifier, clockId denotes a clock identifier, chanId denotes a channel identifier, actId denotes an action, modeId denotes a mode identifier, pred denotes a predicate, expr denotes an expression and type denotes a static type. It is not allowed to use a dot ('.') in an identifier. An expression can also contain a tuple of expressions, syntactically denoted by a comma separated list of expressions surrounded by round braces, for example $(1 + 3, 2)$. Similar to an expression, a variable (clock) identifier can contain a tuple (x, y) of variable (clock) identifiers x and y.

4.1 Input and output variables

Input variables and output variables are special classes of external variable declarations. The semantics ensures that, under the assumption that an automaton does not restrict the behavior of its own input variables, the input variables can change arbitrarily in action and time transitions when that automaton is composed in parallel with another automaton, as long as the inputs are unconnected, or connected to other input variables only. In this way, when input variables are connected to an output variable, the behavior of the connection is completely determined by the output variable. Output variables cannot be connected to output variables of parallel automata. An output variable can be connected to an output variable of an outer block.

4.2 Closed and open scopes

The building blocks that support hierarchy and modularity are the closed scope ⟦ *cScopeDecls* :: *automaton* ⟧ and open scope |(*oScopeDecls* :: *automaton*)|. Here, the double colon acts as a separator between the declarations of variables, action labels and channels on the one hand, and the specification of the behavior on the other hand. Declarations can be internal (intern keyword) or external (extern, input or output keyword). The main difference between closed and open scopes is that open scopes may have *free* variables, that is variables that are not bound in the open scope itself, but in an outer, more global, encompassing scope. A closed scope, on the other hand, may not contain free variables: all variables must be bound in the closed scope.

Both concepts of scoping can be found in modeling languages. The concept of open scopes is often used in different syntactic forms, such as the "for loop" used in many programming and modeling languages. It uses a local iterator that can be used in the body of the for loop together with the variables that are declared at a more global level. This local iterator is essentially a local variable declared in an open scope in which the body of the for loop is executed. The concept of closed scopes can be found in many modeling languages, such as Modelica (see Modelica Association [2002]), where interfaces specify the interaction points (ports) of components explicitly, and connectors are used to connect these ports. The ports and connectors can be mapped to the external variables and connect sets, respectively, of the CIF.

When scopes are nested, inner scopes may declare variables using a name (identifier) that is already used in a variable declaration in an outer scope. A variable used in an automaton binds to the declaration of that variable in the smallest enclosing scope that declares the variable. The search for this variable declaration starts in the smallest enclosing scope of the automaton. If the variable declaration is not found in an open scope, the search proceeds one level up in the scope hierarchy. If it is not found in a closed scope, the model is erroneous, because closed scopes may not have free variables.

By means of connect sets, variables from different(!) closed automata (BNF non-terminal *cAutomaton*) may be connected such that the connected variables refer to the same variable. The following connections are possible:

- Hierarchical connections: The internal and external variables of a closed automaton can be connected to external variables of the contained closed automata.
- Parallel connections: In a parallel composition of closed automata, the external variables of each closed automaton can be connected to external variables of the other closed automata.

Variables x and y are connected if they both occur in a connect set $\{x, y\}$ or if they occur in different connect sets for which the intersection is non-empty, for example $\{x, z\}$, $\{y, z\}$. Within a connect set declaration, it is not allowed that two or more variables that are declared in the same scope are connected. Action labels and channels can be connected to other action labels and channels, respectively, in a similar way as connections between variables are specified.

Section 5 presents several examples of the use of open and closed scopes, together with examples of the use of connect sets, internal and external variables and channels.

5. EXAMPLE: BOTTLE FILLING SYSTEM

The bottle filling system as shown in Figure 1 consists of a liquid storage tank, two identical bottle filling lines, and a bottle supply (see Man and Schiffelers [2006]).

The bottles are filled with liquid from the storage tank. A control system keeps the volume V_T in the storage tank between 2 and 10, and the pH level (acidity) of the liquid in the storage tank between 7 and 7.1. The liquid in the storage tank slowly becomes less acidic (pH level increases). To correct this, a strong acid is dribbled into the storage tank when the acidity of the liquid becomes too low ($pH \geq 7.1$).

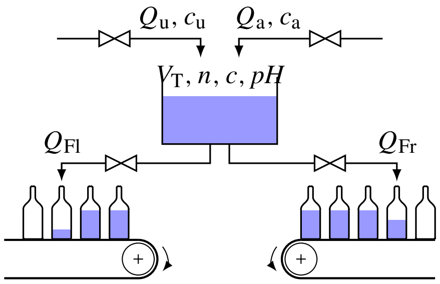

Fig. 1. The bottle filling system.

The acid and liquid supply processes are not modeled, since we consider the acid and liquid always to be available, and we are not interested in the amount of acid or liquid that is used.

The storage tank and the two bottle filling lines are connected by means of the variables Q_{Fl}, and Q_{Fr}, respectively. The volume of the storage tank is available in both bottle filling lines (variable V_{T}) to prevent filling of the bottles when the storage tank is empty.

The molar quantity and molar concentration of the acid in the storage tank are denoted by n and c, respectively, where $n = cV$. The incoming flows of liquid and acid of the liquid storage tank T are denoted by Q_{u} and Q_{a}, respectively. Acid leaves the tank in outgoing flows Q_{Fl} and Q_{Fr}. The gradual reduction of the acidity of the liquid is modeled by means of a constant K_{loss}, which leads to $\dot{n} = c_{\mathrm{u}}Q_{\mathrm{u}} + c_{\mathrm{a}}Q_{\mathrm{a}} - cQ_{\mathrm{Fl}} - cQ_{\mathrm{Fr}} - K_{\mathrm{loss}}V$, where c_{u} and c_{a} denote the concentrations of acid in the flows Q_{u} and Q_{a}. Taking into account that the units of c are in [mol/m^3] instead of [mol/l], the pH is given by $pH = -\log c/1000$.

In the CIF specification of the liquid storage tank, symbols `Qseta`, `Qsetu`, `ca`, `cu`, and `Kloss` denote constants. The behavior of the liquid storage tank is explained as follows. Initially, the `pH` of the liquid in the storage tank equals 7. It is assumed that the pH level of the incoming liquid is 7 or more, since the acidity controller can only make the acidity of the storage tank increase, causing the pH to decrease. If the pH value exceeds the maximum value (`pH >= 7.1`), the acid valve is opened (`alpha:=1`) so that acid is dribbled into the tank. Dribbling of the acid continues until the pH value comes back at 7, and the valve is closed (`alpha:=0`). In a similar way, the controller tries to keep the level of the storage tank between 2 and 10.

The behavior of the filling controller is explained as follows. When a new crate of bottles arrives, (`bottles?n`), where `n` denotes the number of bottles in a crate) the bottle volume is reset to 0, and filling a bottle is started (`(VB,gamma) := (0,1)`). The valve switching the flow `QF` is modeled by means of the discrete variable `gamma`. Filling stops when the volume in the storage tank drops below 0.5 (`when VT<=0.5 now do gamma:=0`). Filling resumes when the volume in the storage tank is at least 0.7. Filling also stops when the bottle is full (`when VB>=1 now do (gamma,n) := (0,n-1)`).

```
model Bottle_Filling_System =
|[ connect {tank.V, left.VT, right.VT}
         , {tank.QF, left.QFl}
         , {tank.QF, right.QFr}
         , {bs.bottles, left.bottles, right.bottles}
:: tank:
     |[ output var V: cont real
      , extern var QFl, QFr: alg real
      , intern var alpha, beta: disc nat = (0,0)
                 , n: cont real
                 , pH: alg real = 7
                 , c, Qa, Qu: alg real
     :: |( mode physics =
               inv dot V = Qu + Qa - QFl - QFr
                 & dot n = cu*Qu + ca*Qa - c*QFl
                           - c*QFr - Kloss*V
                 &     n = c*V
                 &    pH = - log c/1000
                 &    Qa = alpha*Qseta
                 &    Qu = beta*Qsetu
        :: physics
        )|
     || |( mode closed = when pH >= 7.1
                         now do alpha := 1 goto opened
              , opened = when pH <= 7
                         now do alpha := 0 goto closed
        :: closed
        )|
     || |( mode closed = when V <= 2
                         now do beta := 1 goto opened
              , opened = when V >= 10
                         now do beta := 0 goto closed
        :: closed
        )|
     ]|
|| left : Bottle_Filling_Line
|| right : Bottle_Filling_Line
|| bs : Bottle_Supply
]|

automaton Bottle_Filling_Line =
|[ input var VT: real
 , extern var QF: alg real
        , chan bottles?: nat
 , connect {fp.gamma, fc.gamma},
           {fp.VB, fc.VB},
           {bottles, fc.bottles}
:: fp : Filling_Physics
|| fc : Filling_Controller
]|

automaton Filling_Physics =
|[ input var gamma: nat
 , output var VB: cont real
 , extern var QF: alg real
:: |( mode m = inv dot VB = QF
                 &     QF = gamma*QsetF
   :: m
   )|
]|

automaton Filling_Controller =
|[ input var VB, VT: real
 , output var gamma: disc nat = 0
 , extern chan bottles?: nat
 , intern var n: disc nat = 0
:: |( mode start =
             when n = 0 act bottles?n
               do (VB,gamma) := (0,1) goto filling
             when n > 0
               now do (VB,gamma) := (0,1) goto filling
         , filling =
             when VT <= 0.5
               now do gamma := 0 goto stopped,
             when VB >= 1
               now do (gamma,n) := (0,n-1) goto start
         , stopped =
             when VT >= 0.7
```

```
              now do gamma := 1 goto filling
   :: start
   )|
]|

automaton Bottle_Supply =
|[ extern chan bottles!: nat
 , intern clock t
:: |( mode m = when t >= 2 act bottles!24
                do t:= 0 goto m
   :: m
   )|
]|
```

Figure 2 shows a graphical representation of (a part of) the CIF model of the bottle filling line. Solid (dashed) boxes represent closed (open) scopes, where the internal declarations are listed in the upper left corner, and the external declarations are represented as boxes (for variables) or triangles (for channels) on the borders of the box. Modes are visualized by means of circles, urgent (nonurgent) edges are represented as double (single) arrows between modes, and labelled with their guard, action, and update.

6. MAPPING CONCRETE TO ABSTRACT SYNTAX

This section defines the formal semantics of a CIF model specified in concrete syntax by means of a mapping to the abstract format. First the concrete syntax is preprocessed as described in Section 6.1. The mapping of the concrete syntax to the abstract syntax is defined by means of function $\mathcal{T}$. It takes the preprocessed CIF model as input, and returns an automaton specified in the abstract syntax. This function is defined in Section 6.2.

6.1 Preprocessing

First, declarations of the form input var *varIds* : type are replaced by extern var *varIds* : alg type and declarations of the form output var *varDecls* are replaced by extern var *varDecls*. All closed scopes, open scopes and automaton instantiations that occur in the body of the top-level closed scope which are not prefixed with an automaton identifier are prefixed with a fresh identifier.

Using a bottom-up approach, each closed scope prefixed with an automaton identifier is replaced with the same closed scope in which all variables, clocks, channels and actions (including those in nested scopes) are prefixed with the automaton identifier and a dot ('.'). The automaton identifier is removed from the closed scope. A similar approach is used for replacing open scopes and automaton instantiations. In this way, all variables, clocks, channels and actions occurring in the body of the top-level closed scope are made unique. The external variable declarations are grouped w.r.t. their dynamic type into three different lists. The lists $discvars_{\mathrm{ext}}$, $contvars_{\mathrm{ext}}$, and $algvars_{\mathrm{ext}}$ denote a comma seperated list of external variables declared with dynamic type disc, cont, and alg, respectively. Each entry in such a list is of the form $x : t = e$, where x denotes a variable, t denotes its static type, and e denotes its initial value if the initial value is specified at the declaration. The declared external channels are grouped in list $chans_{\mathrm{ext}}$, where each entry in the list is either of the form $c\ !\ : t$, $c\ ?\ : t$, or $c : t$, where c denotes a channel , and t denotes its type. Notation $\overline{x_0\ :\ t_0 = e_0, \ldots, x_n\ :\ t_n = e_n}$ denotes the list $x_0, \ldots, x_n$, notation $\overline{\{x_0\ :\ t_0 = e_0, \ldots, x_n\ :\ t_n = e_n\}}$ denotes the set $\{x_0, \ldots, x_n\}$, and notation $\overline{c_0 \Diamond_0 t_0, \ldots, c_n \Diamond_n t_n}$, where $\Diamond_i \in \{!\ :, ?\ :, :\}$ for $i \in \{0, \ldots, n\}$ denotes denotes the list $c_0, \ldots, c_n$. The clocks and actions are grouped into respective lists $clocks_{\mathrm{ext}}$, and $acts_{\mathrm{ext}}$. A similar notation is used for the internal declarations. Connect set declarations are combined into new connect set declarations that specify the same connections between the identifiers such that each identifier occurs at most in one connect set (for example the connect set declaration connect $\{x, z\}, \{y, z\}$ is replaced by connect $\{x, y, z\}$. Finally, all automaton instantiations occurring in the model are flattened using the automaton definitions of the specification: Let

automaton $l(p_0 : t_0, \ldots p_n : t_n) =$
$[\![$ extern var $discvars_{\mathrm{ext}}, contvars_{\mathrm{ext}}, algvars_{\mathrm{ext}}$
, clock $clocks_{\mathrm{ext}}$, chan $chans_{\mathrm{ext}}$, act $acts_{\mathrm{ext}}$
, intern var $discvars_{\mathrm{int}}, contvars_{\mathrm{int}}, algvars_{\mathrm{int}}$
, clock $clocks_{\mathrm{int}}$, chan $chans_{\mathrm{int}}$, act $acts_{\mathrm{int}}$
, connect $set_1, \ldots, set_h$
:: *automaton*
$]\!]$

be the automaton definition associated with automaton definition identifier l, then automaton instantiation $l(e_0, \ldots e_n)$ denotes the following automaton:

$|($ extern var $discvars_{\mathrm{ext}}, contvars_{\mathrm{ext}}, algvars_{\mathrm{ext}}$
, clock $clocks_{\mathrm{ext}}$, chan $chans_{\mathrm{ext}}$, act $acts_{\mathrm{ext}}$
, intern var $discvars_{\mathrm{int}}, p_0 : t_0 = e_0, \ldots, p_n : t_n = e_n,$
, $contvars_{\mathrm{int}}, algvars_{\mathrm{int}}$
, clock $clocks_{\mathrm{int}}$, chan $chans_{\mathrm{int}}$, act $acts_{\mathrm{int}}$
, connect $set_1, \ldots, set_h$
:: *automaton*
$)|$

The parameter variables $p_0, \ldots, p_n$ of the automaton definition are declared as internal discrete variables inside the closed scope automaton, with initial values $e_0, \ldots, e_n$. Then, the possibly incomplete atomic automaton constructs that occur in *automaton* are made complete according to the following defaults/transformations:

- an omitted invariant denotes the invariant true,
- an omitted flow denotes the flow true,
- an omitted tcp predicate denotes the tcp predicate true,
- an omitted guard denotes the guard true,
- for each urgent edge, the time can progress predicate of the source mode is augmented with the disjunction of negation the guard of this edge. Then, the keyword now is removed from the edge.
- an omitted label denotes the (non-synchronizing) label τ,
- an update assignment $x := e$ is replaced by $\{x\} : x = e^-$, an omitted update denotes the empty update $\emptyset$: true, i.e. the values of the discrete and continuous variables and the clocks are not changed.

6.2 Function $\mathcal{T}$

Model The translation of a specification is defined as follows:
$$\mathcal{T}(\mathsf{model}\ m = closedScope) = \mathsf{encap}_{\mathcal{L}_{\mathrm{com}}}(\mathcal{T}'_{(\emptyset,\emptyset,\emptyset,\emptyset,\mathrm{true})}(closedScope)).$$
Function $\mathcal{T}'$ takes two parameters: the first parameter contains variables, the dynamic type of variables, clocks, actions, and an initialization predicate over variables that are defined at a higher level than the second parameter. The second parameter contains elements of the preprocessed concrete syntax.

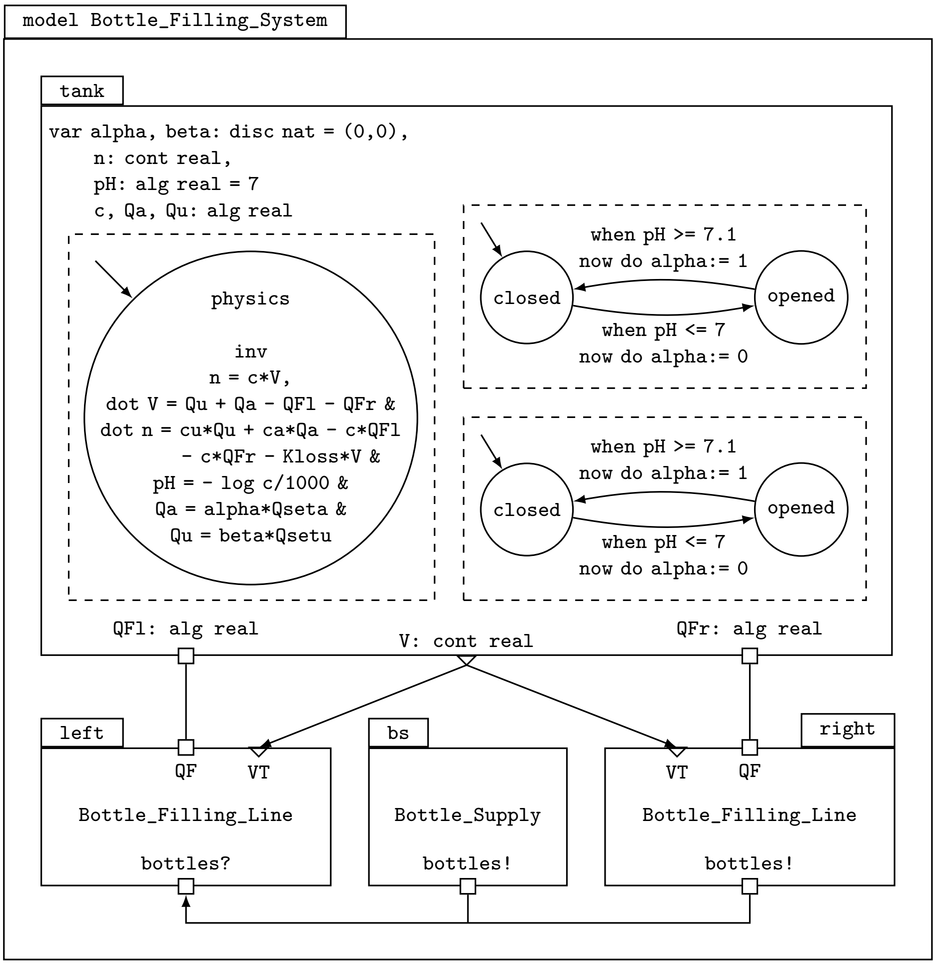

Fig. 2. Graphical representation of the CIF model representing the bottle filling system.

Closed scope A closed scope (BNF non-terminal *closedScope*) is mapped to an abstract CIF automaton as follows:

$$
\begin{aligned}
&\mathcal{T}'_{env}(\llbracket\ \text{extern var } discvars_{\text{ext}}, contvars_{\text{ext}}, algvars_{\text{ext}}\\
&\qquad ,\ \text{clock } clocks_{\text{ext}}, \text{chan } chans_{\text{ext}}, \text{act } acts_{\text{ext}}\\
&\qquad ,\ \text{intern var } discvars_{\text{int}}, contvars_{\text{int}}, algvars_{\text{int}}\\
&\qquad ,\ \text{clock } clocks_{\text{int}}, \text{chan } chans_{\text{int}}, \text{act } acts_{\text{int}}\\
&\qquad ,\ \text{connect } set_1, \ldots, set_h\\
&\qquad ::\ automaton\\
&\qquad \rrbracket\ ) =\\
&\text{hidevar}_{\mathcal{V}\setminus\overline{vars_{\text{ext}}}}(\\
&\quad \text{hideact}_{(\mathcal{L}_{\text{basic}}\cup\mathcal{L}_{\text{com}})\setminus(\{acts_{\text{ext}}\}\cup chanacts_{\text{ext}})}(\\
&\qquad \text{encap}_{\mathcal{L}_{\text{com}}\setminus chanacts_{\text{ext}}}(\mathcal{T}_{env'}(automaton'))), \sigma_{\text{h}})
\end{aligned}
$$

where

- $vars_{\text{ext}} = \{discvars_{\text{ext}}, contvars_{\text{ext}}, algvars_{\text{ext}}\}$,
- $vars_{\text{int}} = \{discvars_{\text{int}}, contvars_{\text{int}}, algvars_{\text{int}}\}$,
- $chanacts_{\text{ext}} = \{h!cs, h?cs, h!?cs \mid h \in \{\overline{chans_{\text{ext}}}\}, cs \in \Lambda^*\}$
- $env' = (\overline{vars}, dtype, acts, clocks, init)$,
- $vars = vars_{\text{ext}} \cup vars_{\text{int}}$,
- $dtype = \{x \mapsto \text{disc} \mid x \in \{\overline{discvars_{\text{ext}}}, \overline{discvars_{\text{int}}}\}\} \cup \{x \mapsto \text{cont} \mid x \in \{\overline{contvars_{\text{ext}}}, \overline{contvars_{\text{int}}}\}\} \cup \{x \mapsto \text{alg} \mid x \in \{\overline{algvars_{\text{ext}}}, \overline{algvars_{\text{int}}}\}\}$,
- $acts = \{acts_{\text{ext}}\} \cup \{acts_{\text{int}}\}$,
- $clocks = \{clocks_{\text{ext}}\} \cup \{clocks_{\text{int}}\}$,
- $init = \wedge_{x:\, x\in\overline{vars}, \text{value}_{vars}(x)\neq\perp} : x = \text{value}_{vars}(x)$,
- $automaton' = automaton[Id_1, \ldots, Id_h/set_1, \ldots, set_h]$,
- $\text{dom}(\sigma_{\text{h}}) = \emptyset$,

where function application $\text{value}_{vars}(x)$ returns the initial value for variable x that is specified in $vars$ or $\perp$ otherwise. Notation $automaton[Id_1, \ldots, Id_h/set_1, \ldots, set_h]$ denotes the automaton where all occurrences of identifiers from set_i in $automaton$ are replaced with identifier Id_i for $i \in \{1, \ldots, h\}$. Identifier Id_i is defined as $Id_i \in set_i \cap (\overline{vars} \cup acts \cup \{\overline{chans_{\text{ext}}}\} \cup \{\overline{chans_{\text{int}}}\})$ if $set_i \cap (\overline{vars} \cup acts \cup \{\overline{chans_{\text{ext}}}\} \cup \{\overline{chans_{\text{int}}}\}) \neq \emptyset$, and $freshId_i$ otherwise, where $freshId_i$ denotes a fresh identifier.

Note that for a closed scope, the environment env is irrelevant, and its elements are not used in the automaton. An automaton can be a closed scope, an atomic automaton, an open scope, or a parallel composition of automata. In the next subsections, the mapping of the latter three is described.

Atomic automaton An atomic automaton (BNF non-terminal *atomicAut*), is mapped to an abstract CIF automaton as follows:

$$
\begin{aligned}
&\mathcal{T}'_{(vars,dtype,acts,clocks,init)}(\\
&\llparenthesis\ \text{init } init_{\text{aut}}\\
&,\ \text{mode} V_1 = \text{inv } i_1 \text{ flow } f_1 \text{ tcp } u_1\\
&\qquad\qquad \text{when } g_{1_1} \text{ act } a_{1_1} \text{ do } up_{1_1} \text{ goto } V_{1_1}\\
&\qquad\qquad \vdots\\
&\qquad\qquad \text{when } g_{1_{k_1}} \text{ act } a_{1_{k_1}} \text{ do } up_{1_{k_1}} \text{ goto } V_{1_{k_1}}\\
&\vdots
\end{aligned}
$$

$$
\begin{aligned}
&, \ \mathsf{mode} V_n = \mathsf{inv}\ i_n \mathsf{flow}\ f_n \mathsf{tcp}\ \ u_n \\
&\qquad\qquad \mathsf{when}\ g_{n_1}\ \mathsf{act}\ a_{n_1}\ \mathsf{do}\ up_{n_1}\ \mathsf{goto}\ V_{n_1} \\
&\qquad\qquad \vdots \\
&\qquad\qquad \mathsf{when}\ g_{n_{k_n}}\ \mathsf{act}\ a_{n_{k_n}}\ \mathsf{do}\ up_{n_{k_n}}\ \mathsf{goto}\ V_{n_{k_n}} \\
&:: v_0 \\
&)|) = (X, \emptyset, dtype, V, v_0, \mathsf{init}, \mathsf{flow}, \mathsf{inv}, \mathsf{tcp}, acts, E)
\end{aligned}
$$

where

- $X = vars \cup clocks$,
- $V = \{V_1, \ldots, V_n\}$,
- $\mathsf{init} = init \wedge init_{\mathrm{aut}} \wedge (\wedge_{x : x \in clocks} : x = 0)$,
- $\mathsf{dom}(\mathsf{flow}) = \mathsf{dom}(\mathsf{inv}) = \mathsf{dom}(\mathsf{tcp}) = V$, $\forall_{i : i \in \{1,\ldots,n\}} : \mathsf{flow}(V_i) = f_i \wedge (\wedge_{x : x \in clocks} : \dot{x} = 1)$, $\mathsf{inv}(V_i) = i_i$, $\mathsf{tcp}(V_i) = u_i$,
- $E = \{(V_i, g_{i_j}, a_{i_j}, up_{i_j}, V_{i_j}) \mid i \in \{1, \ldots, n\}, j \in \{1, \ldots, k_i\}\}$.

Open scope An open scope (BNF non-terminal *openScope*) is mapped to an abstract CIF automaton as follows: Let $env = (vars, dtype, acts, clocks, init)$, then

$$
\begin{aligned}
&\mathcal{T}'_{env}(|(\ \mathsf{extern\ var}\ discvars_{\mathrm{ext}}, contvars_{\mathrm{ext}}, algvars_{\mathrm{ext}} \\
&\qquad , \mathsf{clock}\ clocks_{\mathrm{ext}}, \mathsf{chan}\ chans_{\mathrm{ext}}, \mathsf{act}\ acts_{\mathrm{ext}} \\
&\qquad , \mathsf{intern\ var}\ discvars_{\mathrm{int}}, contvars_{\mathrm{int}}, algvars_{\mathrm{int}} \\
&\qquad , \mathsf{clock}\ clocks_{\mathrm{int}}, \mathsf{chan}\ chans_{\mathrm{int}}, \mathsf{act}\ acts_{\mathrm{int}} \\
&\qquad , \mathsf{connect}\ set_1, \ldots, set_h \\
&\qquad :: automaton \\
&\qquad)|\ ) = \\
&\mathsf{hidevar}_{\overline{vars_{\mathrm{int}}}}(\mathsf{hideact}_{\{acts_{\mathrm{int}}\} \cup chanacts_{\mathrm{int}}}(\\
&\qquad \mathsf{encap}_{chanacts_{\mathrm{int}}}(\mathcal{T}_{env'}(automaton'))), \sigma_{\mathrm{h}})
\end{aligned}
$$

where

- $vars_{\mathrm{int}} = \{discvars_{\mathrm{int}}, contvars_{\mathrm{int}}, algvars_{\mathrm{int}}\}$,
- $chanacts_{\mathrm{int}} = \{h!cs, h?cs, h!?cs \mid h \in \overline{\{chans_{\mathrm{int}}\}}, cs \in \Lambda^*\}$
- $env' = (vars', dtype', acts', clocks', init')$,
- $vars' = vars \cup \overline{vars_{\mathrm{int}}}$,
- $dtype' = dtype \cup \{x \mapsto \mathsf{disc} \mid x \in \overline{\{discvars_{\mathrm{int}}\}}\} \cup \{x \mapsto \mathsf{cont} \mid x \in \overline{\{contvars_{\mathrm{int}}\}}\} \cup \{x \mapsto \mathsf{alg} \mid x \in \overline{\{algvars_{\mathrm{int}}\}}\}$,
- $acts' = acts \cup \{acts_{\mathrm{int}}\}$,
- $clocks' = clocks \cup \{clocks_{\mathrm{int}}\}$,
- $init = init \wedge (\wedge_{x :\ x \in \overline{vars}_{\mathrm{int}}, \mathsf{value}_{vars_{\mathrm{int}}}(x) \neq \perp} : x = \mathsf{value}_{vars_{\mathrm{int}}}(x))$,
- $automaton' = automaton[Id_1, \ldots, Id_h / set_1, \ldots, set_h]$,
- $\mathsf{dom}(\sigma_{\mathrm{h}}) = \emptyset$.

Parallel composition Function $\mathcal{T}_{env}$ distributes over parallel composition: $\mathcal{T}_{env}(automaton_{\mathrm{l}} \parallel automaton_{\mathrm{r}}) = \mathcal{T}_{env}(automaton_{\mathrm{l}}) \parallel \mathcal{T}_{env}(automaton_{\mathrm{r}})$

7. CONCLUDING REMARKS

The presented concrete format consists of the major building blocks required for hybrid system specification. Future work entails, among others, a further concretization of the syntax, including the definition of compound data types, and the definition of the syntax of expressions and equations; extending the concrete format with urgent actions, urgent channels and OR-super states; and possibly extending the interchange format with stochastic model primitives. The development of translations to and from other languages and simulator implementations will be done by different partners in a) Work Package 3 of the HYCON NoE (see HYCON Network of Excellence [2005]), and in b) the new FP7 STREPS MULTIFORM project.

REFERENCES

D. A. van Beek, K. L. Man, M. A. Reniers, J. E. Rooda, and R. R. H. Schiffelers. Syntax and consistent equation semantics of hybrid Chi. *Journal of Logic and Algebraic Programming*, 68(1-2):129–210, 2006.

D. A. van Beek, M. A. Reniers, J. E. Rooda, and R. R. H. Schiffelers. Foundations of an interchange format for hybrid systems. In Alberto Bemporad, Antonio Bicchi, and Giorgio Butazzo, editors, *Hybrid Systems: Computation and Control, 10th International Workshop*, volume 4416 of *Lecture Notes in Computer Science*, pages 587–600, Pisa, 2007a. Springer-Verlag.

D. A. van Beek, M. A. Reniers, J. E. Rooda, and R. R. H. Schiffelers. Revised hybrid system interchange format. Technical Report HYCON Deliverable D3.6.3, HYCON NoE, 2007b.

Stefano Di Cairano, Alberto Bemporad, and Michal Kvasnica. An architecture for data interchange of switched linear systems. Technical Report D 3.3.1, HYCON NoE, 2006.

G. Frehse, O. Stursberg, S. Engell, R. Huuck, and B. Lukoschus. Verification of hybrid controlled processing systems based on decomposition and deduction. In *2001 IEEE International Symposium on Intelligent Control*, pages 150–155, Mexico City, 2001. IEEE.

Goran Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control, 8th International Workshop*, volume 3414 of *Lecture Notes in Computer Science*, pages 258–273. Springer-Verlag, 2005.

Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Decomposing refinement proofs using assume-guarantee reasoning. In Ellen Sentovich, editor, *2000 IEEE/ACM International Conference on Computer-Aided Design*, pages 245–252, San Jose, California, 2000. IEEE.

C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood-Cliffs, 1985.

HYCON Network of Excellence. http://www.ist-hycon.org/, 2005.

Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.

K. L. Man and R. R. H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. PhD thesis, Eindhoven University of Technology, 2006.

MoBIES team. HSIF semantics. Technical report, University of Pennsylvania, 2002. internal document.

Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling*. http://www.modelica.org, 2002.

Alessandro Pinto, Luca P. Carloni, Roberto Passerone, and Alberto L. Sangiovanni-Vincentelli. Interchange format for hybrid systems: Abstract semantics. In João P. Hespanha and Ashish Tiwari, editors, *Hybrid Systems: Computation and Control, 9th International Workshop*, volume 3927 of *Lecture Notes in Computer Science*, pages 491–506, Santa Barbara, 2006. Springer-Verlag.

The MathWorks, Inc. *Using Simulink, version 6*. http://www.mathworks.com, 2005.