Transferring Causality Analysis from Synchronous Programs to Hybrid Programs

Kerstin Bauer and Klaus Schneider
Department of Computer Science
University of Kaiserslautern
{k_bauer,klaus.schneider}@cs.uni-kl.de

Abstract

Outputs of synchronous programs may suffer from cyclic dependencies since statements are allowed to read the current outputs' values to determine the actions that generate the current values of the outputs. For this reason, compilers have to perform a causality analysis that ensures that at any point of time, there is a unique and constructive way to determine the outputs. The discrete parts of hybrid systems may suffer from the same problem as observed in synchronous programs. As we recently extended our synchronous language Quartz to describe hybrid systems, we explain in this paper how the causality analysis as originally introduced for synchronous systems can also be used to handle cyclic dependencies in hybrid Modelica programs.

1 Introduction

Reactive systems [20] are systems that have an ongoing interaction with their environment in terms of a discrete sequence of reaction steps. In each reaction step, all the current inputs are read to compute the outputs for the current point in time as well as the system's state for the next point of time.

Synchronous languages [6, 18, 7] such as Esterel [8, 10], Lustre [19], and Quartz [31] have been developed to describe reactive systems. The operational semantics of these languages is defined by so-called *micro and macro steps*, where a macro step consists of finitely many micro steps whose maximal number is known at compile time. Macro steps correspond to reaction steps of the reactive system, and micro steps correspond to atomic actions like assignments of the program. Variables of a synchronous program are *synchronously updated* between macro steps, so that the execution of the micro steps of one macro step is done in the same variable environment of their macro step.

The distinction between micro and macro steps does not only lead to a convenient programming model for reactive systems that allows to efficiently *synthesize hardware and software* as well as a simplified estimation of worst-case reaction times. It is also the key to a *compositional formal semantics* which is a necessary requirement for formal verification and provably correct synthesis procedures. Typically, the semantics are described by means of SOS (structural operational semantics [27]) transition rules that recursively follow the syntax of the program [9, 31].

As synchronous programs are allowed to read their own outputs, mutual dependencies between trigger conditions and the effect of their actions can occur which is also well-known in hardware design [35, 24, 34, 29, 30]. The causality problem is the problem to decide whether such cyclic dependencies can be resolved at runtime for all reachable states and all possible inputs. It is moreover required that for all inputs and all reachable states, there must be a schedule to fire the enabled actions in a sequential schedule so that all values that were required are available when the actions need them. The mere existence of a unique solution of the cyclic equation

systems is thereby not sufficient, the solution must be determined in a constructive way that can be found at runtime. One therefore often speaks of constructive programs [9] that are based on Brouwer's intuitionistic/constructive logic.

In contrast to the discrete reaction steps of an embedded reactive system, its *environment often consists of continuous behaviors* that are determined by the laws of physics. For this reason, the verification of properties that depend on the interaction with the continuous environment requires the consideration of *hybrid systems* (see e.g. [2, 1]). Since most verification problems are undecidable for hybrid systems [22, 21], a *rich theory* of algorithmic, approximative approaches has been developed over the years [28, 25, 23, 16, 17, 3, 13]. However, *only a few languages and tools* that deal with non-trivial hybrid systems [14] are available. Moreover, the languages of most of these tools focus purely on the continuous part of the system providing only little support for modeling and analyzing of the discrete parts of the system [12].

The Modelica language [15, 26] is a highly developed language for the modeling and simulation of hybrid systems. Academic and industrial tools for handling Modelica descriptions together with a vast amount of libraries are available, e.g., the commercial tool Dymola[1] and the academical/industrial tool OpenModelica[2]. However, as it is the case for the other tools, the emphasis of these tools lies on the modeling of the continuous part of the system. Causality cycles in the discrete part of Modelica programs, called algebraic loops, can currently not be handled by these tools.

In this paper, we therefore propose that causality cycles (i.e. algebraic loops) in Modelica programs can be resolved in the same way as done for synchronous programs: By means of a three- or four-valued logic one may perform a causality analysis at compile time that can assure that during runtime, all the cycles can be resolved [32, 33]. To this end, we report about the experiences we made by extending our synchronous programming language Quartz [31] to describe hybrid systems [4, 5]. We have implemented a simulator for the hybrid Quartz language as well as a translation to Modelica programs. We thereby observed that our compiler was able to deal well with causality cycles, while these programs were rejected by tools that are based on Modelica. We see no problem by extending these tools so that they can also benefit from the solutions that are already successfully used for synchronous languages.

2 The Synchronous Language Quartz

Quartz [31] is a synchronous language derived from the Esterel language [11, 8]. The common paradigm of synchronous languages is the perfect synchrony [18, 7] which means that the execution of programs is divided into macro steps that may be interpreted as logical time. As this logical time is the same in all concurrent threads, these threads run in lockstep, which leads to a deterministic form of concurrency. Macro steps are divided into finitely many micro steps that are atomic actions of the programs. Moreover, variables change synchronously in macro steps, i.e., variables have unique values in each macro step.

In the following, we only give a brief overview of Quartz, and refer to [31] for further details. Provided that S, S_1, and S_2 are statements, ℓ is a location variable, x is a variable, σ is a Boolean expression, and ρ is a type, then the following are statements (parts given in square brackets are optional):

- `nothing` (empty statement)
- $x = \tau$ and $\text{next}(x) = \tau$ (assignments)

---

[1]http://www.dymola.com
[2]http://www.openModelica.org

- $\mathtt{assume}(\varphi)$ and $\mathtt{assert}(\varphi)$ (assumptions and assertions)
- $\ell:\mathtt{pause}$ (start/end of macro step)
- $\mathtt{if}\ (\sigma)\ S_1\ \mathtt{else}\ S_2$ (conditional)
- $S_1;S_2$ (sequences)
- $\mathtt{do}\ S\ \mathtt{while}(\sigma)$ (loops)
- $S_1 \parallel S_2$ (synchronous concurrency)
- $\mathtt{[weak]\ [immediate]\ abort}\ S\ \mathtt{when}(\sigma)$
- $\mathtt{[weak]\ [immediate]\ suspend}\ S\ \mathtt{when}(\sigma)$
- $\{\rho\ x;\ S\}$ (local variable x of type ρ)

The pause statement defines a control flow location ℓ – a boolean variable being true iff the control flow is currently at $\ell:\mathtt{pause}$. Since all other statements are executed in zero time, the control flow only rests at these positions in the program, and thus the possible control flow states are the subsets of the set of locations.

There are two variants of assignments that both evaluate the right-hand side τ in the current macro step (variable environment). While immediate assignments $x = \tau$ immediately transfer the value of τ to the left-hand side x, delayed assignments $\mathtt{next}(x) = \tau$ transfer this value only in the following step.

If the value of a variable is not determined by assignments, a default value is computed according to the declaration of the variable. To this end, declarations consist of a *storage class in addition to the type* of a variable. There are two storage classes, namely $\mathtt{mem}$ and $\mathtt{event}$ that choose the previous value ($\mathtt{mem}$ variables) or a default value ($\mathtt{event}$ variables) in case no assignment determines the value of a variable. Available data types are booleans, bitvectors, signed/unsigned bounded/unbounded integers, arrays and tuples.

In addition to the statements known from other imperative languages (conditionals, sequences and loops), Quartz offers synchronous concurrency $S_1 \parallel S_2$ and sophisticated preemption and suspension statements, as well as many more statements like generic statements to allow comfortable descriptions of reactive systems (see [31] for the complete syntax and semantics).

Our Averest system[3] provides algorithms that translate a synchronous program to a set of guarded actions [31], i.e., pairs (γ, α) consisting of a trigger condition γ and an action α. Actions are thereby assignments $x = \tau$ and $\mathtt{next}(x) = \tau$, assumptions $\mathtt{assume}(\varphi)$, or assertions $\mathtt{assert}(\varphi)$. The meaning of a guarded action is obvious: in every macro step, all actions are executed whose guards are true. Thus, it is straightforward to construct a symbolic representation of the transition relation in terms of the guarded actions (see [31]).

3 Causality Analysis in System Modeling of Controllers

3.1 Causality Problems in Quartz

As already mentioned, synchronous programs often suffer from cyclic dependencies since the programs are allowed to read their own outputs for determining these outputs. It is simple to determine whether a program has such cyclic dependencies by means of a static syntactic analysis. If no cycles occur, it is straightforward to generate code that exactly implements the given program's behavior. However, if cycles occur, the programs can get stuck in deadlocks or may implement one of many possible nondeterministic behaviors. For this reason, a causality analysis has to be performed that checks whether the cycles can be resolved during runtime for all possible states and inputs.

To explain the causality analysis performed in compilers for synchronous languages, consider the execution of a Quartz program. To this end, we assume that
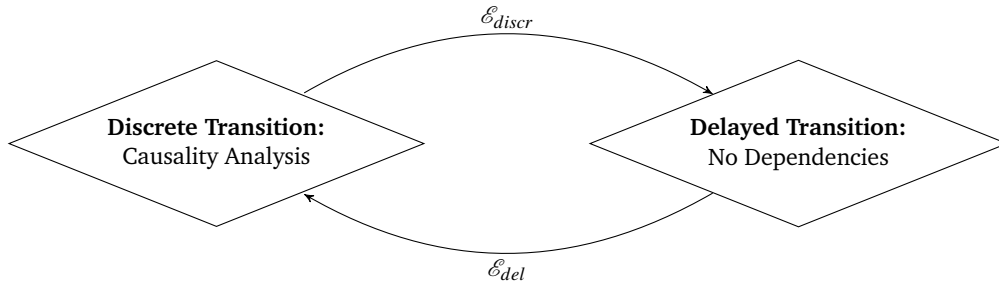
---

[3]http://www.averest.org

Figure 1: Execution of a Macro Step of Quartz

program	guarded actions
| ```
module P(event ?i,o1,o2) {
    if(i) {
        if(o1) o2=true;
    } else {
        if(o2) o1=true;
    }
    assert(!o1 & !o2);
}
``` | ```
regular guarded actions:
     i & o1 ==> o2=true;
    !i & o2 ==> o2=true;
    true    ==> assert(!o1 & !o2);

reaction to absence actions:
  !(i & o1)  ==> o2=false;
  !(!i & o2) ==> o2=false;
``` |

Figure 2: Example Quartz Program with a Causality Cycle and its Translation to Guarded Actions.

the program has already been compiled in a set of guarded actions (γ, α) as explained in the previous section. The execution of such a set of guarded actions is then best explained by a discrete two-location automaton as shown in Figure 1. Each macro step starts with a partial environment $\mathscr{E}_{del}$ that is determined by the delayed actions of the previous macro step[4]. After assigning this partial environment to the current discrete environment $\mathscr{E}_{discr}$, it is checked which trigger conditions become true so that further actions can be executed. Also, actions whose trigger condition become false, can be singled out. Due to the execution of new actions, more values become known, and the same procedure is repeated until no more values become known. The program is constructive (i.e. causally correct) if the environment $\mathscr{E}_{discr}$ is fully defined at the end. Finally, the delayed actions are executed to provide a new delayed environment $\mathscr{E}_{del}$ for the next execution step. As all values are known at that point of time, the computation of $\mathscr{E}_{del}$ does not require a causality analysis.

A simulator can directly implement the procedure outlined above. A compiler has to perform this analysis for all inputs, which is typically done in a symbolic way (similar to model-checking) to avoid the enumeration of all states and inputs. Although it is possible to remove the causality cycles if the programs are constructive, it is often better to retain them in the code, since it is known that the cyclic code will be typically smaller [24, 29, 30] and will not have problems during runtime (if the code is executed as outlined above).

A simple example for a Quartz program with cyclic dependencies is shown in Figure 2. The program P has a boolean input i and two boolean outputs o1 and o2 that can also be read. As o1 and o2 are boolean event variables, they are reset to their default value false whenever there is no action setting them actively.

The translation to guarded actions yields the guarded actions shown on the right of Figure 2. We distinguish between the 'regular' guarded actions that are those that appear in the program, and additional guarded actions that are added by the compiler to reset event variables or to store memorized variables.

---

[4]In the first macro step initial assignments are given instead.

Iteration Step	i	o1	o2	May-Set of Actions	Must-Set of Actions
1	`true`	$\perp$	$\perp$	$o1 \Rightarrow o2 = \text{true}$ $\neg o1 \Rightarrow o2 = \text{false}$ $\neg o2 \Rightarrow o1 = \text{false}$	$\text{true} \Rightarrow o1 = \text{false}$
2	`true`	`false`	$\perp$		$\text{true} \Rightarrow o2 = \text{false}$ $\text{true} \Rightarrow o1 = \text{false}$
3	`true`	`false`	`false`		$\text{true} \Rightarrow o2 = \text{false}$ $\text{true} \Rightarrow o1 = \text{false}$

Table 1: Fixpoint Iteration for Input Value `i = true`

The causality analysis for input `i=true` is shown in Table 1. It can be seen that the values of all variables can be determined within three steps even though `o1` and `o2` depend on each other. Depending on the so-far obtained partial variable environment, the set of guarded actions is partitioned into 'must actions' that must be executed (their trigger condition is true), 'cannot actions' that cannot be executed (their trigger condition is false), and the remaining 'may actions' whose might or might not be executed (their trigger condition is neither true nor false due to still unknown variables). The value $\perp$ shown in Table indicates that currently no valid value is known for the corresponding variable.

A similar fixpoint iteration can be shown for `i = false`. However, in this case, the value of `o2` will be computed before the one of `o1`, so that a different schedule has to be used depending on input `i`.

```
model causality
    Real t    (start = 0);
    Boolean i  (start = false);
    Boolean o1 (start = false);
    Boolean o2 (start = false);
equation
    der(t) = 1.0;
    when { t>=0.1 } then
        i = if pre(i) then false else true;
        o1 = if (not(i) and o2) then true else false;
        o2 = if (i and o1) then true else false;
        reinit(t,0.0);
    end when;
end causality;
```

Figure 3: Modelica Program with Causality Conflict

3.2 Causality Problems in Modelica

The Modelica language also supports the synchronous model of computation in its discrete part. Because of this, the guarded actions obtained from Quartz programs can be easily translated to equivalent Modelica programs. However, the tools we used were not able to deal with causality problems (algebraic loops), illustrated with the Modelica program shown in Figure 3. This program is obtained by translating the guarded actions of the Quartz program P to Modelica. The real variable *t* is only a time trigger and does not influence the discrete behavior of the program.

Neither OpenModelica 1.5.0 nor the demoversion of the industrial tool Dymola 7 are able to execute the program, because of the occurring algebraic loop. Even when simplifying the line `i = if pre(i) then false else true;` by `i = true;` such that there exists a unique computation order as shown in Table 1, the tools cannot handle the program. Failure reports of Dymola simply note: "Current version cannot generate code for an algebraic loop involving integers or Boolean".
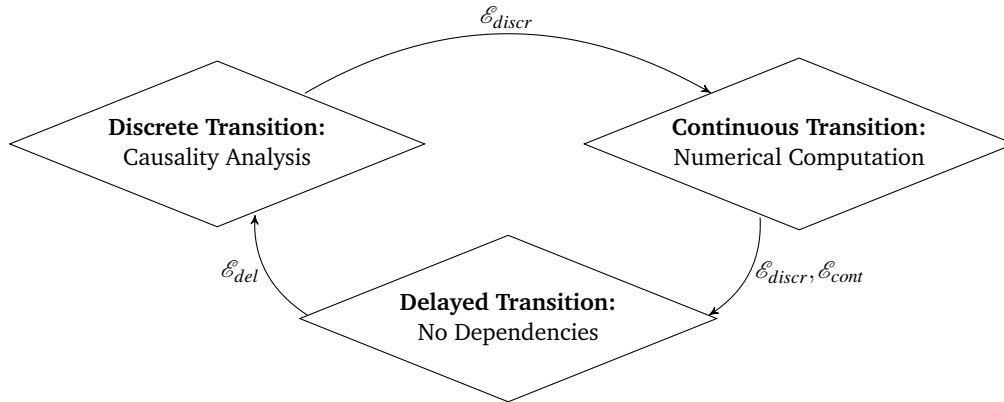
Figure 4: Execution of a Macro Step of Hybrid Quartz

4 Hybrid Quartz

The environments of embedded reactive systems are often defined by continuous behaviors that are determined by the laws of physics. To be able to describe these continuous environments, the synchronous language Quartz has been recently extended to Hybrid Quartz [4]. While time in synchronous languages is purely logical, hybrid systems require the consideration of physical time. In order to combine these inherently different time concepts, the computational model of macro steps is endowed by a continuous transition as depicted in Figure 4.

In addition to the memorized and event variables, there are additionally hybrid variables in Hybrid Quartz. Hybrid variables must have the data type `real` and are the only variables who have continuous evolutions during the macro steps. These continuous variables can be assigned by *flow assignments* x <- τ whose left hand side may also refer to the derivation of the continuous variable drv(x) <- τ. For specification and verification reasons, one can additionally impose constraints constrain{S,M,E}(ϕ), which state that the condition ϕ has to be satisfied at the **S**tart, at any inter**M**ediate point or at the **E**nd of the continuous evolution.

The continuous actions x <- τ, drv(x) <- τ, and constrain{S,M,E}(ϕ) may only occur in special statements of the form flow S until(σ) where S is a list of flow assignments and σ is a so-called *release condition* that terminates the continuous phase defined by the flow statement. The compiler also generates guarded actions for Hybrid Quartz, where the actions now additionally consist of continuous guarded actions, i.e., guarded actions that contain flow assignments or constrain actions as well as guarded actions $(\gamma, \texttt{release}(\sigma))$ for the release conditions.

In Hybrid Quartz, there exists also a new operator `cont` that allows one to access the discrete as well as the continuous value of a hybrid variable during a flow-statement, i.e., its value at the time when the continuous phase was started and its value at some considered point of time during the continuous phase.

More information on hybrid Quartz can be found in [4].

The simulation of Hybrid Quartz programs, depicted in Figure 4, is performed as follows: After computing the discrete variable environment $\mathcal{E}_{discr}$ by means of the causality analysis, one can determine which of the continuous actions are enabled. Taking the values of $\mathcal{E}_{discr}$ as initial values for potential systems of ordinary differential equations, the continuous flow of the macro step is executed until the first active release condition σ is satisfied. $\mathcal{E}_{cont}$ stores the values of all variables at the end of the continuous transition. Note, that $\mathcal{E}_{discr}$ and $\mathcal{E}_{cont}$ coincide in all discrete variables, as these do not change their value during the continuous evolution. The delayed transition now obtains both environments as inputs and computes the partial environment $\mathcal{E}_{del}$ for the following macro step.
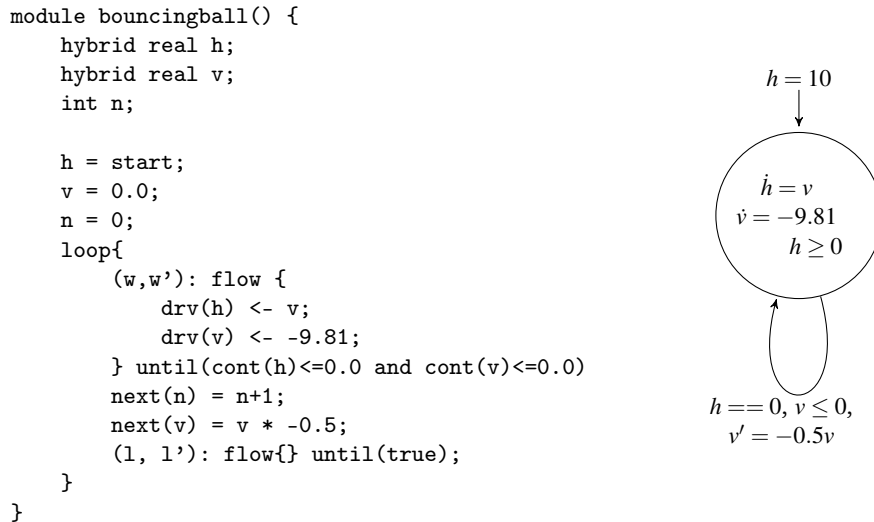
```
module bouncingball() {
    hybrid real h;
    hybrid real v;
    int n;

    h = start;
    v = 0.0;
    n = 0;
    loop{
        (w,w'): flow {
            drv(h) <- v;
            drv(v) <- -9.81;
        } until(cont(h)<=0.0 and cont(v)<=0.0)
        next(n) = n+1;
        next(v) = v * -0.5;
        (l, l'): flow{} until(true);
    }
}
```

$h = 10$

$$\dot{h} = v$$
$$\dot{v} = -9.81$$
$$h \geq 0$$

$$h == 0, v \leq 0,$$
$$v' = -0.5v$$

Figure 5: Bouncing Ball Example in Hybrid Quartz

We conclude this section by listing a Quartz program for the well-known Bouncing Ball example in Figure 5. The program models a bouncing ball, whose velocity is reduced by half whenever it bounces on the floor. Initially, the ball starts at height h = 0.0 with velocity v = 0.0, and no bounces appeared so far (n = 0).

5   From Hybrid Quartz to Modelica Programs

The intermediate code format of Hybrid Quartz in the form of guarded actions is already very close to Modelica code. Only few adjustments need to be made in order to obtain an equivalent Modelica program. The translated Modelica program will have the general form

```
model fromQuartz
    Variable Declarations;
equation
    all continuous guarded actions
    when (disjunction of all guarded release conditions)
        for each variable: gather all guarded actions in one expression
        for each continuous variable: reinit with the associated discrete value
    end when;
end fromQuartz:
```

Hybrid Quartz essentially provides two environments for each macro step, $\mathscr{E}_{discr}$ and $\mathscr{E}_{cont}$. The third environment $\mathscr{E}_{del}$ can be omitted in Modelica models, as one can use the pre-operator instead.

To obtain an equivalent Modelica model, it is necessary to represent these two environments in some way. The easiest way is to create for each variable a discrete version and a continuous version, where the discrete version refers to $\mathscr{E}_{discr}$ and the continuous version refers to $\mathscr{E}_{cont}$. As the two environments coincide on all discrete variables, it suffices to only create a copy for each hybrid variable. That means, each hybrid variable hybrid real x in Quartz is replaced in Modelica by

```
discrete Real x_discr; Real x;
```

whereas each discrete[5] variable type y in Quartz is replaced by

```
discrete type y;.
```

---

[5]The storage class of discrete variables is handled by the compiler in that appropriate reactions to absence are generated.

The guarded actions are translated into Modelica in two successive steps: In the first step, each guarded action $\gamma \Rightarrow \alpha$ is translated into the equivalent statement

$$\texttt{if } \sigma \texttt{ then } \alpha.$$

As this code is not yet supported by the existing tools, these guarded actions will be gathered together in actions of the form

$$\texttt{x = if } \gamma_1 \texttt{ then } \alpha_1 \texttt{else if } \dots$$

which are accepted by both Dymola and OpenModelica.

The first type of actions are immediate assignments in the form of $\gamma \Rightarrow \texttt{x } =\tau$. This action is replaced by

$$\texttt{if } \gamma_{discr} \texttt{ then x = } \tau_{discr}$$

where γ_{discr}, τ_{discr} replace each occurrence of a continuous variable x by either x or x_discr, depending on whether x lies within a cont(_) statement or not.

The second type of guarded actions is that of delayed actions $\gamma \Rightarrow \texttt{next(x) = } \tau$. Analogously to the discrete actions, first γ and τ are replaced by γ_{discr} and τ_{discr}. In a second step, all variables x are replaced by pre(x) in γ_{discr} as well as in τ_{discr}, as the computation of these actions must be done w.r.t. the variable environment of the previous macro step. This has the same effect as computing the delayed assignments in the current macro step and storing them in an intermediate environment.

Expressions within flow actions conditions are treated analogously.

In the second step, for each variable all conditional assignments are collected and written as a single assignment with the variable on the left hand side. Additionally, according to the storage class of the variable, the reaction to absence is encoded:

- event variables are set to their default value.
- memorized variables are set to their previous discrete value.
- hybrid variables are set to their last known continuous value.

Equations concerning the continuous flow are written first in the equation setting. As condition for the following when-statement, the boolean disjunction of all release actions is given.

According to simple test models, Modelica or at least the demo version of Dymola and OpenModelica are not capable to handle execution steps, where the continuous evolution actually does not consume physical time. Therefore we must add a timer t which is reinitialized to 0 during each discrete transition and increases linearly in time. All activation conditions of the when statement now must be conjuncted with the condition that time has actually advanced, i.e. $t \geq min$, where min is some minimal time (chosen suitable small).

Within the when-statement, now all discrete equations are written down. Furthermore, all hybrid variables are reinitialized with their discrete values.

The translation procedure is finally illustrated by the bouncing ball example given in Figure 5. In a first step, the program is compiled to guarded actions. The boolean variable Init will hold iff the execution is in the initial step. Furthermore, in order to increase readability, the boolean expression $cont(h) <= 0.0$ and $cont(v) <= 0.0$ in the flow statement is replaced by σ. The corresponding Modelica model is given in Figure 6.

```
1. (Init ∨ l )∧σ => next(w) = true
2. (w' ∨ Init ∨ l )∧¬σ => next(w') = true
3. w => next(l) = true
4. Init ∨ l ∨ w' => der(h) <- v
5. Init ∨ l ∨ w' => der(v) <- -9.81
6. w => next(n) = n+1
7. w => next(v) = v*-0.5
8. Init ∨ l ∨ w' => release(σ)
```

```
model BouncingBall
    Real h (start = 10);
    discrete Real h_discr (start = 0);
    Real v;
    discrete Real v_discr (start = 0);
    Boolean Init(start = true);
    Boolean w(start = false);
    Boolean w2(start = false);
    Boolean l  (start = false);
    Integer n (start = 0);
    Real t(start=0);

equation
    der(t) = 1;
    der(h) = if (Init or l or w2) then v else 0;
    der(v) = if (Init or l or w2) then -9.81 else 0;

    when ((t>=0.01) and (((Init or l or w2) and h < 0 and v < 0) or w)) then
            n = if pre(w) then pre(n) + 1 else pre(n);
            v_discr = if pre(w) then  - pre(v)  * 0.5     else pre(v);
            h_discr = pre(h);
            w = if (pre(Init) or pre(l) or pre(w2)) and pre(v) <= 0 and pre(h) <= 0
                    then true else false;
            w2 = if (pre(Init) or pre(l) or pre(w2)) and not (pre(v) <= 0 and pre(h) <= 0)
                    then true else false;
            l     = if pre(w) then true else false;
            Init = false;

            reinit(v, v_discr);
            reinit(h, h_discr);
            reinit(t, 0);
    end when;
end BouncingBall;
```
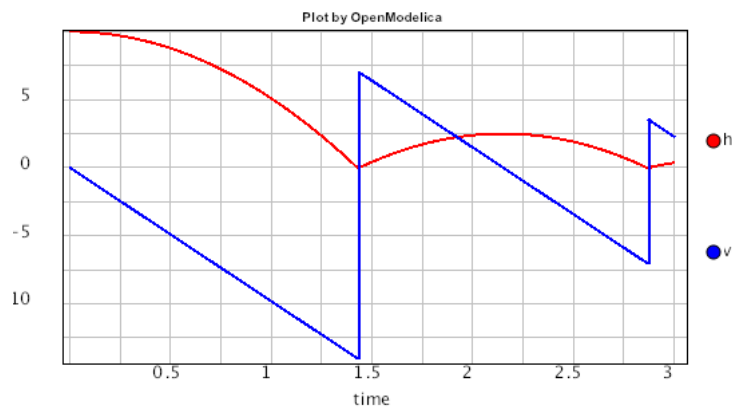

Figure 6: Modelica Model of the Bouncing Ball generated from a Hybrid Quartz Program

6 Conclusions

Hybrid Quartz and Modelica approach the modeling of hybrid systems from different starting points: While Hybrid Quartz emphasizes mainly the discrete part of the system, the Modelica language puts its emphasis on the continuous part. Nevertheless, this paper shows that both languages not only share a common core, but that one can translate Hybrid Quartz programs one-to-one to Modelica programs.

However, due to the different origins of the two modeling languages, Hybrid Quartz and Modelica provide very different algorithms for the analysis and simulation of the hybrid programs: Based on the well-developed theory within the discrete domain, Quartz (like other synchronous languages) is able to solve non-trivial algebraic loops, thus allowing a much broader variety of models. As non-trivial causal dependencies may easily occur, this is a big advantage over Modelica programs. Furthermore, discrete Quartz already provides algorithms for formal verification.

On the other hand, as the main emphasis of Modelica lies on the continuous part of the hybrid system, algorithms to deal with the continuous evolutions of such systems are well-developed. As Quartz only has been recently extended to Hybrid Quartz, sophisticated algorithms for dealing with the continuous dynamics are still missing.

Thus, tools for both languages could learn a lot from each other. Tools for synchronous languages offer sophisticated procedures for formal verification, worst case execution time analysis and causality analysis, while tools that deal with Modelica offer much better support for continuous dynamics. Thus, the resulting mixture could be a lot more powerful as both tools on their own.

References

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science (TCS)*, 138(1):3–34, 1995.

[2] R. Alur, C. Courcoubetis, T. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In R. Grossmann, A. Nerode, A. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *LNCS*, pages 209–229. Springer, 1993.

[3] R. Alur, T. Henzinger, G. Lafferriere, and G. Pappas. Discrete abstractions of hybrid systems. *Proceedings of the IEEE*, 88(7):971–984, 2000.

[4] K. Bauer and K. Schneider. From synchronous programs to symbolic representations of hybrid systems. In K. Johansson and W. Yi, editors, *Hybrid Systems: Computation and Control (HSCC)*, pages 41–50, Stockholm, Sweden, 2010. ACM.

[5] K. Bauer and K. Schneider. Predicting events for the simulation of hybrid systems. In *International Conference on Embedded Software and Systems (ICESS)*, Bradford, United Kingdom, 2010. IEEE Computer Society.

[6] A. Benveniste and G. Berry. The synchronous approach to reactive real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[7] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[8] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

[9] G. Berry. The constructive semantics of pure Esterel. `http://www-sop.inria.fr/esterel.org/`, July 1999.

[10] G. Berry. The Esterel v5 language primer. `http://www-sop.inria.fr/esterel.org/`, July 2000.

[11] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[12] X. Briand and B. Jeannet. Combining control and data abstraction in the verification of hybrid systems. In R. Bloem and P. Schaumont, editors, *Formal Methods and Models for Codesign (MEMOCODE)*, pages 141–150, Cambridge, Massachusetts, USA, 2009. IEEE Computer Society.

[13] D. Campagna and C. Piazza. Hybrid automata in systems biology: How far can we go? *Electronic Notes in Theoretical Computer Science (ENTCS)*, 229:93–108, 2009.

[14] L. Carloni, M. Di Benedetto, R. Passerone, A. Pinto, and A. Sangiovanni-Vincentelli. Modeling techniques, programming languages, and design toolsets for hybrid systems, 2004. Report on the Columbus Project, http://www.columbus.gr.

[15] P. Fritzson and V. Engelson. Modelica - a unified object-oriented language for system modeling and simulation. In *Object-Oriented Programming*, volume 1445 of *LNCS*, pages 67–90. Springer, 1998.

[16] M. Fränzle. What will be eventually true of polynomial hybrid automata? In N. Kobayashi and B. Pierce, editors, *Theoretical Aspects of Computer Software (TACS)*, volume 2215 of *LNCS*, pages 340–359, Sendai, Japan, 2001. Springer.

[17] R. Ghosh, A. Tiwari, and C. Tomlin. Automated symbolic reachability analysis with application to delta-notch signaling automata. In O. Maler and A. Pnueli, editors, *Hybrid Systems: Computation and Control (HSCC)*, volume 2623 of *LNCS*, pages 233–248, Prague, Czech Republic, 2003. Springer.

[18] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.

[19] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[20] D. Harel. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[21] T. Henzinger. The theory of hybrid automata. In *Logic in Computer Science (LICS)*, pages 278–292, New Brunswick, New Jersey, USA, 1996. IEEE Computer Society.

[22] T. Henzinger, P. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? In *Las Vegas, Nevada, USA*, pages 373–382. ACM, 1995.

[23] G. Lafferriere, G. Pappas, and S. Yovine. A new class of decidable hybrid systems. In F. Vaandrager and J. van Schuppen, editors, *Hybrid Systems: Computation and Control (HSCC)*, volume 1569 of *LNCS*, pages 137–151, Berg en Dal, The Netherlands, 1999. Springer.

[24] S. Malik. Analysis of cycle combinational circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, 13(7):950–956, July 1994.

[25] J. Miller. Decidability and complexity results for timed automata and semi-linear hybrid automata. In N. Lynch and B. Krogh, editors, *Hybrid Systems: Computation and Control (HSCC)*, volume 1790 of *LNCS*, pages 296–309, Pittsburgh, Pennsylvania, USA, 2000. Springer.

[26] Modelica Association. Modelica - a unified object-oriented language for physical systems modeling, language specification version 2.0, 2002. http://www.Modelica.org.

[27] G. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Aarhus, Denmark, 1981.

[28] S. Ratschan and Z. She. Safety verification of hybrid systems by constraint propagation based abstraction refinement. In M. Morari and L. Thiele, editors, *Hybrid Systems: Computation and Control (HSCC)*, volume 3414 of *LNCS*, pages 573–589, Zürich, Switzerland, 2005. Springer.

[29] M. Riedel and J. Bruck. Cyclic combinational circuits: Analysis for synthesis. In *International Workshop on Logic and Synthesis (IWLS)*, Laguna Beach, California, USA, 2003.

[30] M. Riedel and J. Bruck. The synthesis of cyclic combinational circuits. In *Design Automation Conference (DAC)*, pages 163–168, Anaheim, California, USA, 2003. ACM.

[31] K. Schneider. The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2009.

[32] K. Schneider and J. Brandt. Performing causality analysis by bounded model checking. In *Application of Concurrency to System Design (ACSD)*, pages 78–87, Xi'an, China, 2008. IEEE Computer Society.

[33] K. Schneider, J. Brandt, T. Schuele, and T. Tuerk. Maximal causality analysis. In J. Desel and Y. Watanabe, editors, *Application of Concurrency to System Design (ACSD)*, pages 106–115, St. Malo, France, 2005. IEEE Computer Society.

[34] T. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *European Design and Test Conference (EDTC)*, Paris, France, 1996. IEEE Computer Society.

[35] L. Stok. False loops through resource sharing. In *International Conference on Computer-Aided Design (ICCAD)*, pages 345–348. IEEE Computer Society, 1992.