Unifying Control and Verification of Cyber-Physical Systems (UnCoVerCPS)

WP4 Tool Support (Task 4.1)

D4.2 – Extension of the Scade language for continuous modeling

Project funded by the European Commission under Horizon 2020, the Framework Programme for Research and Innovation (2014-2020). Grant No.: 643921.

WP6	**D4.2 – Extension of the Scade language for continuous modeling**
Authors	Cédric Pasteur, Jean-Louis Colaço, Goran Frehse
Short Description	This document presents the additional Scade language constructs to support continous behaviors and the interaction between continuous and discrete parts. It also describes the expected code to be generated.
Deliverable Type	R
Dissemination level	PU
Delivery Date	2017/06/12
Contributions by	JL. Colaço, X. Fornari, G. Frehse, C. Pasteur
Internal review by	Dr. J. Oehlerking, Pr. Olaf Stursberg, Dr Mattias Altoff
Date of acceptance	2017/06/12
Keywords	SCADE, hybrid, discrete, continuous, model-based

Document history:

Version	**Date**	**Author/Reviewer**	**Description**
1.0	2017/04/14	Pasteur	Initial version
1.1	2017/05/04	Fornari	Updates from internal review
1.2	2017/05/12	Pasteur/Fornari	Miscellaneous clarifications, conclusion
1.3	2017/05/15	Pasteur/Fornari	Hybrid tools comparizon and wind turbine example.
1.4	2017/06/01	Pasteur/Fornari	Scade vs SpaceEx analysis.
1.5	2017/06/12	Fornari	Final review.

Contents

1 Introduction

This document presents the continous extension of the Scade language, named Scade Hybrid. The main motivation for Scade Hybrid is the capability to describe in a same language continuous and discrete parts while keeping determinism, as explained in section 1.1. Relationship with the UnCoVerCPS project are also explained in this introductory section.

The document is decomposed into the following sections. The section 2 presents the notion of zero-crossing events and their interactions with discrete controllers. The interaction of continuous and discrete parts is realizd by extending state machines, leading to hybrid automata. The hybrid Scade model can be compiled in C code, but the execution has to take care of the interaction between the continuous and discrete parts, which leads to code to execute the discrete part and dedicated code bound to a solver for the continuous part. Introducing continuous capabilities within Scade also allows one integrates continous models coming from other environments into a unified simulation.

Section 3 presents the specifications for the code generator. Scade Hybrid is the result of research done on various prototypes and similar data-flow languages. The work presented here is the final design for the language and the generated code. It presents also the minimum environment necessary as a runtime.

Section 4 describes some experiments done with the language. It presents the integration capability based on a FMU example and the modeling of part of the wind turbine use case (WP5).

The section 5 concludes on the achieved work with respect to the definition and implementation of Scade Hybrid.

In the context of the UnCoVerCPS project, a link between SCADE and SpaceEx will be developed to bring a bridge between verification and implementation. As there are the Scade, Scade Hybrid and SpaceEx formalisms, an analysis is required to compare them. The appendix A provides that analysis and gives directions for a flow combining Scade and SpaceEx. The presented work is an important step to build the UnCoVerCPS toolchain. This work is also connected to WP5 with the automotive use case. The goal will be to analyze the controller environment within SpaceEx and to extract properties that must be fulfilled by the implementation of the controller designed in SCADE.

1.1 Motivations

In the UnCoVerCPS approach, the software development is model-based and code is automatically deployed from SCADE models. The classical V-cycle sofware development phases are

(for development part, not taking into account the validation phases):

1. system requirements, which provide the overall system objectives
2. global specification, which describes the system architecture (functional architecture, allocation of functions, communications) and interfaces with high level behavior of functions;
3. detailed specifications, which give the details of the requirements of the functions. These specifications are often close to final code, as they are given in textual form.
4. code. Depending on the context, there may be more or less constraints on the code. The selected language (object-oriented or not), the coding standard, the traceability means (between specification and code, specification and tests, code and tests), memory management (dynamic or fixed) are one of the possible decisions to take when going for embbeded software. Decisions are taken according the criticality of the application and the confidence one wants to have and one wants to be able to demonstrate;
5. validation, which means testing. Testing ranges from unit testing for functions or even piece of code, to functional testing at system level, and finally testing before actual use in production;
6. verification. Verification is the activity that confirms that the development is done properly. It mainly consists in reviews: review of specifications for accuracy, testability, review of code for standard respect, correctness with respect to specifications, review of tests definitions and results. Formal methods can also be used as verification means.

SCADE is used at the detailed specifications level, where it replaces textual requirements, which is error prone, with a well-defined input. Then the code is automatically generated from that specification and it is guaranteed to fulfill the expected behavior thanks to the certified code generator.

But it is still necessary to perform verification and validation activities at the specification level to ensure its correctness. Using SCADE, one can reduce or eliminate activities like coding or unit testing, and verification of design is alleviated as the language has a clean definition. As functional testing is key, one needs to be able to provide a test environment which is powerful enough to describe the environment but also deterministic to be able to replay the test sessions with the very same results. This is very important in certification domain as the applicant must be able to present the evidences of all his/her activities that he/she did during the complete development.

Therefore being able to mix continous and discrete parts within a deterministic semantics is a real improvement of the current practices. By extending Scade with continuous capabilities it is possible to perform such testing and simulation combining the continuous part and the discrete part, with a well-defined semantics of interactions. This ensures the correctness of the simulation (validation objective) and expected behaviors can be assessed up-front (verification objective). The goal of Scade Hybrid is to increase that level of confidence into the described mixed-models.

As a result, we have the following flow:

1. during system analysis, controller are defined and/or synthetized. Note that at this stage formalisms describing interactions between discrete and continuous worlds are usually non-deterministic, like SpaceEx. This is important to specify the controller behavior in various contexts as non-determinism allows for exploring diverse situations.

2. Scade discrete models can be derived from controller synthesis phase.

3. These models can then be simulated within a model of the environment in Scade Hybrid with guarantees in terms of correctness the interactions and expected results.

4. Finally, certifiable code is generated from the validated model.

This flow permits to develop embeddable controller specifications with full confidence in their correctness, and then obtain certifiable code thanks to the automatic certified code generation.

To be complete, Formal Methods (proof-based methods) can also used to assess the correctness of the controller but no credit can be claimed out of them in industrial high-integrity application context. This is only the case for rail and transportation industry which recommends the use of Formal Methods. Aerospace&Defence recognizes them in the DO-178 standard, but not yet in practice as of today. Automotive is just starting looking at it for autonomous vehicle.

The overall benefits for stakeholders, that is to say companies developing embedded applications and tool providers, is therefore an improved flow with better tool integration, where the transitions between controller design, software specification and implementation are smoother and confidence increase.

This also opens new horizons for customers as they will be able to develop new kind of applications. In particular, there is a trend where a control application can be coupled with an embedded simulation of the process. This is more powerful that monitoring of the safety region where the controller must stay. Indeed, the embedded simulated process allows for prediction of the possible failures of the actually controlled process, with less sensors or with

access to normally inaccessible data. In that perspective, Scade Hybrid is a mean to describe such integration with the control and the process.

1.2 Scade Overview

This section gives a short description of the Scade language. More information on it can be found in [9].

Scade originates from an effort in the 80' to define proper languages to program high-integrity applications. Such applications have two main characteristics: a) they control a process using a *read inputs/compute/write outputs* cycle, b) a defect can have catastrophic effects involving possible fatalities. One of the answer was the synchronous languages family relying on the zero-delay hypothesis. Moreover these languages have a well-defined and unambiguous semantics.

Scade is a data-flow oriented language, close to the applied control engineer practices. Its semantics relies on the Kahn Process Network [8], which is a model of distributed computation. The language is strongly typed and declarative. In 2006, Scade has been extended to natively supports state-machines. As such, it is the only language supporting a full mix of data-flow and control-flow constructs. This extension has been made with two objectives: a) it is conservative, b) safety must still be ensured.

The point a) ensures that the Scade paradigm is preserved and the overall semantics is well-defined, as new constructs are derived from core language notions. As a result of point b), the design of the state-machines privileges clear notions so that a model can be easily understood, even if the behavior is complex. For instance, there transitions cannot cross state borders to reach some deep internal substates, priorities on transition is well-defined, there is the notion of a *weak/strong* transition for a fine-grain control of the activation of the states. A model is checked by the SCADE Suite certified code generator then, if it is correct with respect to the language semantics, C code is generated. The SCADE Suite code generator is itself developed following the rules of the standards (DO-178, ISO 26262, EN 50128, ...). Every artefact produced during its development is accessible to authorities, which can then assess the correctness of the code generator in great details.

Starting from Scade and its certified code generator, Scade Hybrid is therefore a natural extension to achieve the objective of the mix of discrete and continuous worlds with the highest confidence for embedded software.

2 Scade Hybrid Primer

2.1 Overview

The objective of Scade Hybrid is to add to the discrete Scade notation the capabily to support continuous behavior. Therefore, Scade Hybrid is an extension of the Scade language with constructs to define flows using *ordinary differential equations* (ODE). The resulting language is an explicit hybrid systems modeler, like Mathwork' Simulink or National Instruments' LabView. It is different from implicit hybrid systems modelers like Dassault Systemes' Dymola or ANSYS' Simplorer, which use *differential algebraic equations* (DAE).

The language is based on the work of M. Pouzet et al. on the Zelus language [3], with several main ideas:

- The solution of ODEs is handled by an external numerical solver. This allows one uses an off-the-shelf state-of-the-art numerical solver for efficiency and precision of the computation. This requires a dedicated interface to easily connect any solvers. Note that in any cases, solvers can not provide any guarantee of their computations. Indeed our objective is to provide a clean separation between discrete and continuous part, leaving potential non-determinism in the continuous part. This non-determinism can occur from resolution of computation (how close are we to zero) and resolution of time (do two zero-crossing events (see sectionr̃efsec:zerocross actually occur simultaneously?). Research is still on-going to refine this potential non-deterministic behavior and to how reduce it;

- Discrete-time computations are strictly separated from continuous-time computations. The goal is to ensure that the semantics of a model does not depend on the numerical solver used or its time step. Whereas some tools like Simulink already check such properties and display warnings in some cases, we decide to reject programs that don't separate discrete and continuous computations. The behavior of accepted programs is fully deterministic for the discrete part;

- The existing compilation process and infrastructure should be reused as much as possible. Discrete nodes are compiled exactly as in regular Scade. This compilation process and the simulation algorithm is described in details in [6].

- Eventually, interfacing with reachability analysis tools will be done to explore the state space of the continous part.

The work achieved within the task 4.1 is an industrialization effort of the research work. The Zelus language is an academic language, used for various research experiments, while Scade is a recognized industrial language[1].

Zelus has a limited types support (only on integer type while Scade and Scade Hybrid have several integers and floating point types), Scade Hybrid has arrays. The Scade Hybrid language is the introduction of the hybrid concept within the Scade language as an extension preserving the initial language semantics. The introduction of the hybrid concept must be done with care to ensure that all impacts on the existing language constructs are properly managed, like the interaction with state machines or imperative constructs.

Another difference is that the target code generation language is C instead of OCAML for the Zelus academic compiler. The support of Scade Hybrid will be introduced into the existing qualified code generator with the objective is to reuse a maximum of it. This will pave the way for a certification of a Scade Hybrid to C code generator. Also, the code generator must provide traceability information between the input model of and the generated code. The corresponding work means detailed specification of the new language, as detailed specifications of the generated code and of the internal algorithms.

2.2 A first example

Let us introduce a first example of a Scade Hybrid program, that defines a flow `t` equal to the elapsed time since the beginning of the simulation:

```
hybrid time() returns (t:float64 last=0.0)
    der t = 1.0;
```

The example defines a flow `t` with an initial value equal to `0.0` and which derivative is constant and equal to `1.0`. We will call *continuous state* a flow defined by its derivative. Note we use the term *continuous state* to denote a *state variable*. Scade basic-iest notion is the flow being discrete or now continuous, there is no variable in Scade[2] and we also need to distinguish from discrete states.

The **hybrid** keyword is used to introduce an operator containing continuous constructs. On the other hand, the **node** keyword introduces a discrete-time operator and the **function** keyword is used to define combinatorial operators. Continuous constructs, like **der** equations, can only be used in a continuous context, i.e. inside a **hybrid** operator. Discrete constructs,

[1] In 2017, a record track was established of more than one hundred certifiable/qualifiable projects using Scade in Aerospace&Defence, Rail Transportation, ... Most of the recent C-919 airliner software (flight-control, landing-gear, ...) has been realized with Scade.

[2] Scade is inspired from functional languages

like the **pre** or **fby** operators, can only be used in a discrete context, i.e. inside a **node** operator. Similarly, a **hybrid** (resp. **node**) operator can only be called inside another **hybrid** (resp. **node**) operator. Combinatorial functions can be used inside any context, but can only contain combinatorial operators.

2.3 Zero-crossing events

Continuous and discrete parts of a model are linked by *zero-crossing events*[3]. Such events occur when a continuous state crosses zero, that is, when it goes from positive to negative or vice-versa. Detecting zero-crossing events is one of the features of numerical solvers. A discrete computation can only be triggered by a zero-crossing event. It means that the discrete state of a model only changes during zero-crossing events and remains constant during the integration by the numerical solver.

Here is an example of a program that increments a discrete counter every time the input signal `z` crosses zero from negative to positive:

```
node counter() returns (cpt:int32)
    cpt = 1 -> pre cpt + 1;

hybrid hybrid_counter(z:float64) returns (cpt:int32)
    cpt = (activate counter every up z initial default 0)();
```

The **activate** construct tells to execute the `counter` operator each time there is a positive zero-crossing of x. The result of **activate** is the result of the `counter` operator when activated, else the previous value, initialized to 0. The output of this operator is shown in Figure 1. As one can see, the output of a discrete computation, like the `counter` operator, is a piecewise constant flow, that only changes value at zero-crossing events.

During a discrete step, it is also possible to reset the value of a continuous state. A typical example is the simulation of a bouncing ball, as shown in Figure 2:

```
const y0:float64 = 2.0;
const g:float64 = 9.81;

hybrid simple_ball() returns (y:float64 last = y0; y_v:float64 last = 0.0)
let
    der y = y_v;
    activate if down y
    then y_v = - 0.8 * last 'y_v;
```

[3]In the FMI standard, such events are called *state events* and zero-crossing signals are called *event indicators*.

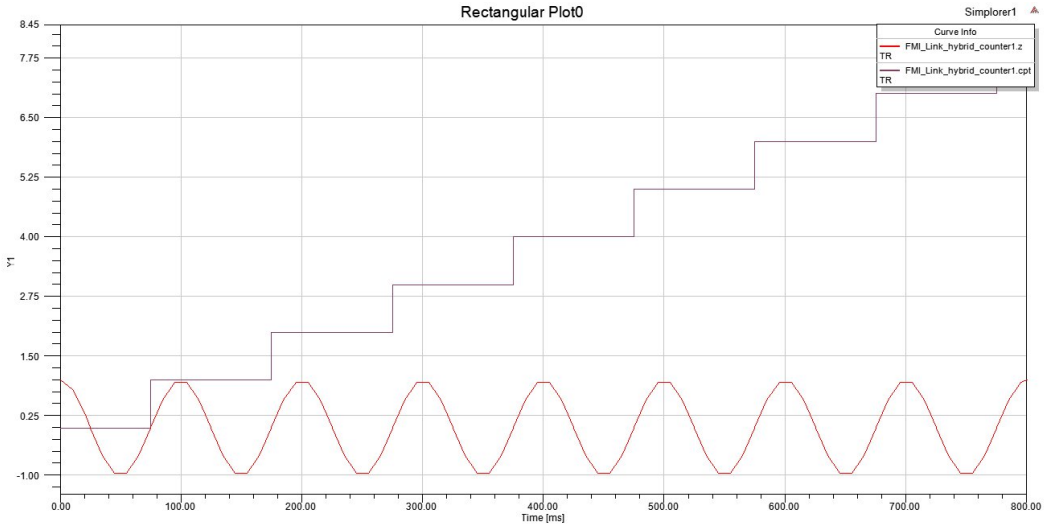


Figure 1: Simulation of the `hybrid_counter` operator

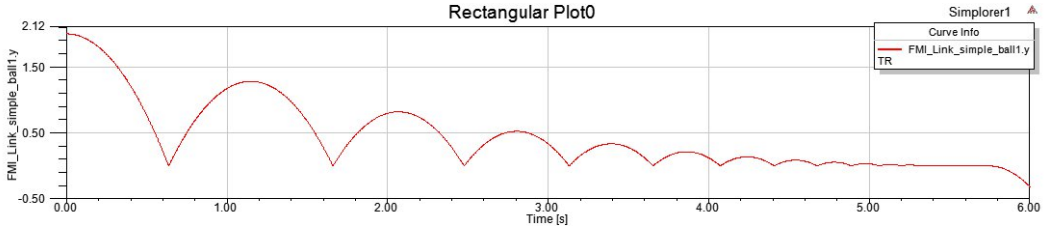


Figure 2: Simulation of the `simple_ball` operator

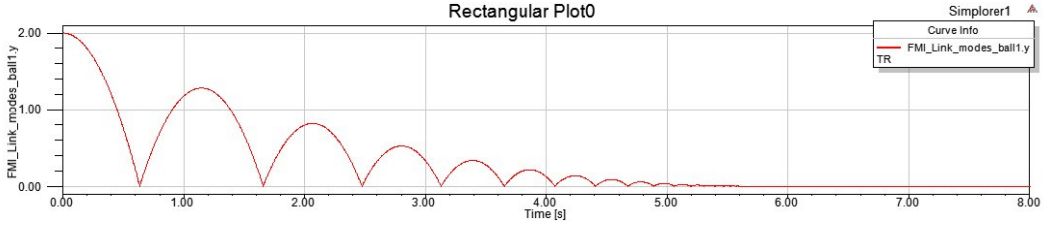


Figure 3: Simulation of the `modes_ball` operator

```
    else der y_v = - g;
    returns y_v;
tel
```

Each time the position `y` of the ball crosses zero, its speed `y_v` is reset to `-0.8` times its previous value, to simulate the contact with the ground[4].

2.4 Hybrid automata

State machines can also be used in a continuous context [2]. In that case, states contain continuous equations, like **der** equations, and transitions are triggered by zero-crossing events, which means that actions on transitions are considered discrete. We can for instance write an improved version of the bouncing ball (Figure 3). In that version we introduce the `modes_ball` automaton.

```
const
    y0 : float64 = 2.0;
    g : float64 = 9.81;
    eps : float64 = 0.001;

hybrid ball() returns (y:float64 last = y0; y_v:float64 last = 0.0)
let
    der y = y_v;
    activate if down y
    then y_v = - 0.8 * last 'y_v;
    else der y_v = - g;
    returns y_v;
tel

hybrid modes_ball() returns (y,y_v:float64)
let
    automaton
        initial state Bouncing
            y, y_v = ball();
        until if down y and y_v < eps restart Sliding;

        state Sliding
        let
```

[4]This example uses the **last** construct which refers to the previous value of an identified flow *declared* in an outer scope. The **pre** refers to the previous value of a flow *expression* used in a local scope

```
            y = 0.0;
            y_v = 0.0;
        tel
    returns y, y_v;
tel
```

In Figure 2, we can see that, after a while, the balls falls below the ground. It is because when the ball gets closer to zero, changing its speed is not enough to make it go above zero. We fix this problem by introducing a two-state automaton:

- In the first state, the ball is bouncing as before. We leave this state when the ball crosses zero and its speed is below a given threshold.
- In the second state, the ball is fixed.

2.5 Time events

A common way to schedule a discrete computation in a continuous setting is to launch it periodically. A periodic event `z` can be created from a continuous state of derivative `1.0` that is reset every `p` seconds:

```
activate if z
then x = 0.0;
else der x = 1.0;
z = up (last 'x - p);
```

There is however a much more efficient way to implement periodic events. Indeed, numerical solvers take as input an *horizon*, which is the maximal date until which the integration should be done. Time events are a built-in feature of the code generation: the horizon given to the numerical solver is the closest deadline of all the operators in the program. Several predefined operators can be used:

- `timer(`*delay*`)` is a one-shot timer that creates an event *delay* seconds later.
- `period(`*off*, *p*`)` creates an event after an offset *off*, then every *p* seconds.

`timer` is an imported operator which just returns the expected horizon. The `period` operator is defined from the `timer` operator:

```
hybrid imported timer(delay:float64) returns (z:zero);

hybrid period(offset, p:float64) returns (z:zero)
let
  automaton
    initial state Init
      z = timer(offset);
    until if z restart Periodic;

    state Periodic
      z = timer(p);
    until if z restart Periodic;
  returns z;
tel
```

More complex behaviors can be implemented using the same ideas.

2.6 Other events

We have already seen two kinds of events that can be used in a continuous context: zero-crossing detection (**up**, **down**, **cross** operators) and time events. These events have type **zero**. Transition or activate blocks expect as condition a flow of type **zero** in a continuous context and type **bool** in a discrete context as usual.

More generally, continuous events are:

- Zero-crossing detection: **up** `z` is active `z` was negative and becomes positive; **down** `z` when it was negative and becomes positive; **cross** `z` in both cases.
- Time events
- A special event denoted **init**, which is active during the first instant of the current continuous context. This discrete instant is for instance used to set the initial value of continuous state. This is equivalent to the expression `timer(0.0)`.
- Composition of events: `z1` **and** `c` is the conjunction of an event and a Boolean condition; `z1` **or** `z2` the disjunction of two events.

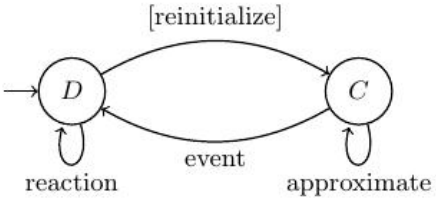


Figure 4: Simulation algorithm

2.7 The `last` operator

In a discrete context, the **last** operator has the same semantics as usual: it returns the value of the flow at the previous discrete step. This definition cannot be used in a continuous context, as this value depends on the method and the time step used by the numerical solver.

As in [1], we thus define **last** as the left limit of a signal. This definition corresponds to the previous one for discrete flows, as they are constant during integration phases. If **last** is used in a discrete context for a continuous signal, it corresponds to the value of the flow right before the event. This value can be computed before triggering the event. In a continuous context, we can only define **last** $\mathbf{'x = x}$ to remain independent of the solver.

In the current prototype, the **last** operator can be used in any context for a continuous state, as their value is an input given by the solver. For other flows, **last** can only be used in a discrete context.

2.8 Execution

The Scade Hybrid code generator generates sequential C code from a model. The execution of a Scade Hybrid program uses a numeric solver as a black box for approximating continuous signals and detecting interesting events with reasonable accuracy

The basic structure of the simulation algorithm is shown in Figure 4. The algorithm begins in an initial discrete phase, D, which initializes both the discrete and the continuous states. After which the algorithm alternates between a continuous phase, C, and a discrete phase, D.

In the C phase, the numeric solver is repeatedly executed to approximate the evolution of continuous states and to search for zero-crossings. The solver in turn makes two types of callbacks: one for the values of continuous state derivatives, and the other for the values of zero-crossing expressions.

The algorithm cycles in the continuous phase, constantly increasing the value of the simulation time, until the solver reports that one or more zero-crossings have been found or the requested horizon has been reached. The algorithm then enters the D phase. The discrete

controller then performs its computation. It produces outputs according to the inputs, the zero-crossing events and its current state and possibly changes its internal state. Once this atomic reaction is finished, the C phase then restarts, and so on.

2.9 Standard library

A Scade Hybrid library called **libhybrid** is distributed with Scade Hybrid. It allows to use the new constructs of the language in SCADE Suite IDE to design graphical Scade Hybrid models. These constructs are defined in a package called **Pervasives**.

The **Hybrid** package defines some commonly used functions, like the integrator (scalar or vector), the unit delay, the PID, etc.

2.10 Scade Hybrid bibliography

Scade Hybrid is based on research work by M. Pouzet et al, which first proposed to extend a synchronous language with primitives to define ODEs [3]. This work was later extended with hierarchical automata [2] similar to the one in Scade. The separation between discrete and continuous is enforced by a simple type system [3]. A causality analysis ensures the absence of causality loops [1]. A semantics based on non-standard analysis was proposed in [4]. The compilation process to statically scheduled code and the execution with an off-the-shelf numerical solver is described in [6]. Theses results are the foundation of Scade Hybrid and of the academic language Zélus[5] [7].

3 Code Generator Specifications

This section describes the specifications of the code generator. The specifications are in two parts. The first part is about the syntax and semantics (in natural language) of the Scade Hybrid extensions of the Scade language. The second part is the description of the generated C code. We describe the C constructs to support new functions, continuous states, zero-crossings events. We also introduce an horizon function that is used to compute the next time at which a continuous computation is expected.

Note that the Zelus academic compiler produces OCAML code. The Scade Hybrid code generator will produces C code, that will be compatible with the existing C code generation from *classical* Scade model. The new C generated code combines the discrete and continous parts. The continuous part is computed by some external solver. This external solver depends on the user environment. Therefore, we propose a generic runtime which role is to perform the

[5] http://zelus.di.ens.fr/

integration between the generated code and the solver. The runtime relies on some predefined C constructs that must be defined in relationship with the solver interface and the actual generated code. An example of automatic generation of the content of the runtime is given for the Functional Mockup Interface standard (see also section 4.2).

3.1 Scade Hybrid specifications

This section describes the Scade Hybrid language. Since Scade Hybrid is an extension of Scade, this section is based on the Scade 6 Reference Manual [9]. Only new constructs are described here. The new constructs are related to:

- new type to handle zero-crossing event;
- new function kind for Scade operators;
- new expressions to support zero-crossing events;
- extension of boolean operation for zero-crossing events;
- modification of higer-order operation to handle zero-crossing events as inputs

3.1.1 Types and groups

type_expr	::=	**bool** | **zero** | ...

A new type called **zero** is introduced. It is the type of for zero-crossing events. Strongly-typed languages like Scade rely on type systems to perform semantics checks and help the user to manipulate proper data type for a given intent. Additionally, a new type kind called **boolean** is introduced to group regular Booleans (**bool**) and zero-crossings (**zero**). Therefore we can discriminate between cases where only event expressions are possible, cases where only Boolean expressions are possible, or cases when it does not matter. In case of incorrect use of an expression in a given context, the code generator will emit a clear message to the user, who can understand the situation and if he/she is modeling properly with respect to the specification needs.

3.1.2 User defined operators

op_kind	::=	**function**
	|	**node**
	|	**hybrid**
where_decl	::=	**where** TYPEVAR {{ , TYPEVAR }} *numeric_kind*
	|	**where** TYPEVAR {{ , TYPEVAR }} **boolean**

A new kind of user-defined operators is defined. These operators are introduced with the keyword **hybrid** and can contain continuous-time equations. It allows to describe a hybrid model modularly, like for discrete model in Scade. The **node** keyword represent discrete nodes and **function** introduces combinatorial functions, that can be used both in a continuous or discrete context.

3.1.3 Equations

equation	::=	...
	|	*der_equation*
der_equation	::=	**der** ID = *expr*

We introduce a new kind of equation to specify an ODE: **der** $x = e$ defines the derivative of x to be equal to e. The initial value of x is given by the **last** value of its declaration. A derivative equation is well typed if the type of the left-hand side matches the type of the right-hand side expression. It can only be used in a continuous context.

The existing control structures of Scade can be used in a continuous context. However, continuous-time and discrete-time equations must be clearly separated:

- In a continuous context, the condition of an if-block must have **zero** type. The **then** branch is then a discrete context, whereas the **else** branch is a continuous context. In a discrete context, the condition of an if-block must have **bool** type.
- In a continuous context, the condition of transitions in a state machine must have **zero** type. The body of each state is a continuous context, whereas the actions on transitions are discrete contexts. In a discrete context, the condition of transitions in a state machine must have **bool** type. The body of each state and the actions on transitions are discrete contexts.
- An emission can only be done in a discrete context.

3.1.4 Expressions

Zero-crossing operators We introduce zero-crossing operators:

expr	::=	*id_expr*
	|	...
	|	*zc_expr*
zc_expr	::=	**up** *expr*
	|	**down** *expr*
	|	**cross** *expr*
	|	**init**

The operators **up**, **down** and **cross** detect the zero-crossing of a signal:

- **up** z is true if z goes from negative to strictly positive
- **down** z is true if z goes from strictly positive to negative
- **cross** z is true if z goes from negative to strictly positive or vice-versa

The **init** keyword is a special event that is true during the first discrete instant where the current continuous context is active. It is equivalent to **true** `->` **false** in a discrete context. It could also be defined as `init = timer(0.0)`.

Boolean operators The type of some Boolean operators are modified:

- **and** takes as first input a value of kind **boolean** (either **bool** or **zero**), as second input a value of type **bool** and returns a value of the same type as the first input.
- **or** takes as inputs and returns values of the same **boolean** type (either **bool** or **zero**).

In the current specification it is not possible to take the conjunction of two zero-crossings. There is still on-going work to provide a definition of what could be simultaneous zero-crossing events knowing that the zero-crossings depend on the solver for the precision of the computed values close to zero and for the precision of the time resolution. Currently, our solution offers an operational semantics, so that we can guarantee that the discrete part is deterministic, but there may be some non-determinism in the continous part. Nevertheless, it is possible to associate Boolean flows with occurences of zero-crossing events and to perform the conjunction on these Boolean flows as a workaround.

Continuous and discrete contexts There are restrictions to the use of operators depending on the context:

- Zero-crossing operators can only be used in a continuous context.
- Discrete operators (like **pre**, **fby**) can only be used in a discrete context.
- Combinatorial operators (eg. arithmetic or Boolean operators) can be used in any context.

Higher-order operators The **activate** higher-order operators can be used with both kind of **boolean**:

- If the condition has type **zero**, the input operator must be discrete (ie. a **node** or **function**) and the resulting expression must be used in a continuous context.
- If the condition has type **bool**, the resulting operator has the same kind as the input operator (either discrete or continuous).

The **restart** operator takes as input a condition of **zero** type in a continuous context and of **bool** type in a discrete context.

3.2 Code generation

To implement the integration of the continuous and the discrete part, we follow the approach described in [3], but applied to our existing tool. The current Scade code generator architecture will be reused, so that the impact will be minimal (only 5% of the code generator should modified). While [3] presents the overall concept and its implementation with OCAML as target language, the new code generator will produce C code and supports more data types.

3.2.1 Generated functions

In the case of a discrete node, we generate as usual:

- An `init` function to initialize the context of the node.
- A `reset` function to reset the internal state of the node.
- A `step` function that computes one step of the node and updates the internal state.

Note that the code generated from a discrete operator is the same as the one generated by KCG² from a (regular) Scade model.

In the case of a **hybrid** operator, we now generate:

- An `init` function to initialize the context of the node.
- A `reset` function to reset the internal state of the node.
- A `step` function that computes one discrete step of the node and updates the internal discrete state. It can read the values of continuous states as computed by the solver and may change them.
- A `cont` function that computes the current value of zero-crossings and derivatives.
- A `horizon` function that returns the next horizon requested by this operator.

3.2.2 Continuous states

To represent derivatives, we introduce a structure type:

```
type kcg_cstate = { last: real; val: real; der: real; };
```

The **real** type is a new internal float type used for continuous states and zero-crossings. It shall be defined as the type used by the numerical solver. Casts are added from/to this type when reading or writing continuous states and zero-crossings. The `kcg_real` type must be defined in a `kcg_hybrid_types.h` header included by `kcg_types.h`.

For each variable defined by its derivative, we add a corresponding field of type **kcg_cstate** in the context. Constructs are translated as follows:

- An equation **der** `x = e` is translated to `x.der = e` in the `cont` function
- An equation `x = e` is translated to `x.val = e` in the `step` function.
- A read of `x` is translated to a read of `x.val`
- A read of **last** `x` is translated to a read of `x.last`

The `x.val` and `x.last` fields are updated by the runtime before the execution of the **step** and **cont** functions. The runtime reads the `x.der` field to get the derivative of each continuous state and the `x.val` field to get the updated value of the continuous state after a discrete step. These values are given to the numerical solver.

3.2.3 Zero-crossings

We proceed the same way for zero-crossings:

```
type kcg_zc = { up: bool; out: real; last:real; };
```

For each zero-crossing `x = up z`, we declare a corresponding field of type **kcg_zc** in the context. Constructs are translated as follows:

- The equation `x = up z` is translated to:

  ```
  x.out = z;
  x = x.up;
  ```

  The first equation is only used in the **cont** method. In the **step** method, `x` is always equal to false. The **last** field of each zero-crossing is copied from the **out** field after each discrete step.

The `x.out` field is read by the runtime and given to the numerical solver to detect zero-crossings. The `x.up` field is updated by the runtime before calling the **step** function to signal the detected events. The `x.last` field is also updated by the runtime.

3.2.4 Horizon computation

The `horizon` method returns the next date at which an event should occur (or `kcg_infinity` if there is no active timer). It is one of the inputs expected by the numerical solver and can be used to efficiently implement time events (see Section 2.5).

Note that there is no construct in the language to set the horizon returned by an operator. Only the imported `timer` operator sets an horizon. Other operators just compute the minimum of the horizons of called operators.

The `horizon` method returns the minimum of the results of calling the corresponding method for all instances of called nodes, including in particular instances of the `timer` operator.

```
kcg_real n_horizon(outC_Root *outC)
{
  kcg_real horizon_acc;
  kcg_real tmp;

  horizon_acc = kcg_infinity;
  tmp = period_horizon(&outC->Context);
  horizon_acc = kcg_min(horizon_acc, tmp);
  return horizon_acc;
}
```

After reading a local horizon, it has to be set to `kcg_infinity`. This ensures that if we call the horizon of an operator that is not active during a step, it will always return `kcg_infinity`.

3.3 Scade Hybrid runtime

The code generated by KCG Hybrid compiler is independent of the backend used. The following backends have been implemented:

- Model-exchange FMU (FMI 1.0 and 2.0): it uses the solver and zero-crossing detection provided by the FMU host.
- Co-simulation FMU (FMI 1.0 and 2.0): it uses the CVode numerical solver[6] to solve ODEs.
- Standalone executable: it is also based on CVode and acts as the simulation master.

3.3.1 Structure of the runtime

The runtime is located in the `lib/` directory:

- `kcg_hybrid_runtime.h` defines generic data structures used by all backends.
- `kcg_hybrid_runtime.c` defines generic functions used by all backends. For instance, the `discrete_steps` function execute discrete steps until the returned horizon is not zero.
- `libhybrid.scade`, `timer.c` and `timer.h` contain the predefined operators for time events (see section 2.5).
- `fmu_generator.py` is a Python script that generates the files used by the several backends (see Section 4.2.1).
- `pack_fmi.py` is a Python script used to build an FMU from a Scade Hybrid model. It takes care of calling the code generator, the `fmu_generator.py` script to generate the FMU files, compiling the generated code and packing the FMU.
- `me_fmu/`: FMI 1.0 and 2.0 model-exchange FMU backend.
- `cs_fmu/`: FMI 1.0 and 2.0 co-simulation FMU backend.
- `cvode_standalone`: the standalone backend using CVode.

[6] https://computation.llnl.gov/projects/sundials/cvode

- `kcg_ext_component.h`, `cs[12]_fmu.[ch]`, `me[12]_fmu.[ch]` and `fmu_wrapper.py` are used for importing FMUs (see Section 4.2.2).

3.3.2 Generic parts of the runtime

The runtime expects two additional files to be generated for each model. These files are generated by the `fmu_generator.py` script since they can be generated using only the information in the mapping file. This allows to minimize the impact on the compiler.

The first file is called `kcg_info.h` and should define several macros about the root operator:

```
/* information used for corresponding FMI attribute */
#define KCG_GUID "8daf05a3-247a-11e5-aa15-2beff3ac55df"
#define MODEL_IDENTIFIER simple_ball

/* root operator functions */
#define STEP_FUN simple_ball
#define RESET_FUN simple_ball_reset
#define INIT_FUN simple_ball_init
#define CONT_FUN simple_ball_cont
#define HORIZON_FUN simple_ball_horizon

/* context and output structure */
#define SCADE_OUT_CTX outC_simple_ball
#define SCADE_IN_CTX inC_simple_ball

/* number of continuous states */
#define NB_CSTATE 2
/* number of zero-crossings */
#define NB_ZC 1
```

The runtime also uses global arrays containing information about input/output variables, continuous states and zero-crossings:

```
/* Information about an input/output */
typedef struct {
  size_t offset;
} var_info;

extern const var_info var_infos[];
```

```
/* Information about a continuous state */
typedef struct {
  size_t offset;
} cstate_info;

#if NB_CSTATE > 0
extern const cstate_info cstate_infos[NB_CSTATE];
#endif

/* Information about a zero-crossing */
typedef enum { UP, DOWN, CROSS } zero_dir;

typedef struct {
  size_t offset;
  zero_dir dir;
} zero_info;

#if NB_ZC > 0
extern const zero_info zero_infos[NB_ZC];
#endif
```

In order to access variables stored in the context, we store its offset so that they can be directly read or updated. This allows for instance to read the derivatives (set by **der** `x = e` equations) to send them to the numerical solver. The arrays storing this information are declared in a file generated by the wrapper generator.

Each backend must declare two header files:

- `kcg_logger.h` must declare macros `TRACE`, `WARNING`, `ERROR` used for error reporting;
- `kcg_hybrid_types.h` which must declare the `kcg_real` type used for storing continuous states and zero-crossing signals;

4 Experiments

This section details experiments done with the Scade Hybrid prototype. The first experiment is a modeling of the Wind Turbine case. The second experiment is a support of the FMI standard.

4.1 Wind Turbine

A model of a wind turbine as a hybrid system was initiallly proposed by GE. This model has been captured in a simplified Scade Hybrid model (not all equations are given), as shown in figure 5. The interface of the automaton is as follows:

inputs `Omg`, for Ω_g, is the rotor speed;

outputs `Mg`, for M_g, is the generator torque and `teta_c`, for θ_c, for the demanded pitch angle.

The obtained hybrid automaton is composed of three states:

- an initial state;
- a state *Region12* computing M_g, from various conditions. Note that in that state θ_c is set to a constant value θ^{opt} in the initial model. This is specified as the default value for the `teta_c` input. Therefore, the equation does not appear in the state;
- a state *Region3* which computes an other value for M_g and computes also the derivative of θ_c. In textual Scade Hybrid, this can be written as **der** `theta_c = f_x_y;`. As the new textual constructs are not yet graphically supported we re-use existing capabilities with a pseudo-integrator. The specific operator is recognized by the code generator which in turn produces the right derivative operation.

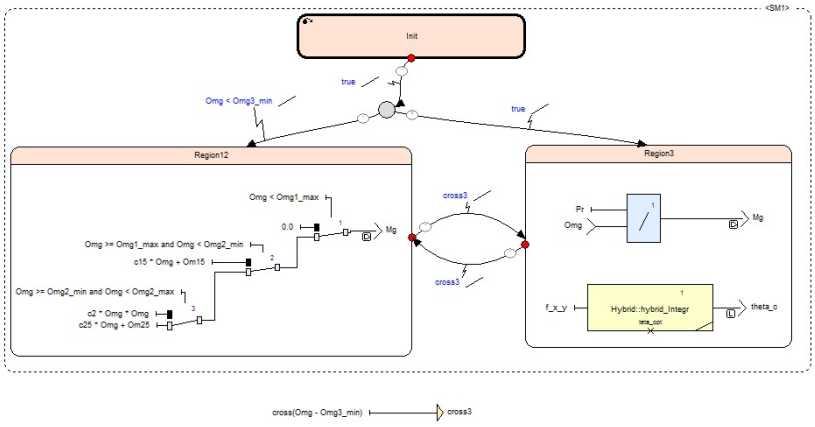


Figure 5: Hybrid model of wind turbine

In parallel to these three states, the graphical equation corresponding to `cross3 = cross(Omg-Om3_min)` which is true whenever the guarded expression becomes 0. `cross3` is used as the condition to switch back-and-forth the two states containing the equations. This shows the interaction between continuous and discrete worlds *via* the zero-crossing detection thanks to the **cross** operator, and the direct interaction between data-flow

and state-flow constructs through the `cross3` bolean data-flow. The example also shows the capability of the Scade language to freely mix data-flow and control-flow constructs. Extensions on that example are the finalizeation the implementation and the execution of a complete simulation using Scade Hybrid.

4.2 FMU code generation experiments

FMI (Functional Mock-up Interface) is a standard[7] designed by the Modelica community. The objective of FMI is to support both model exchange and co-simulation of dynamic models using a combination of xml-files and compiled C-code. Simulation models can be exchanged, while preserving intellectual property if needed (source code is optional). A simulation model is called a Functional Mock-up Unit (FMU). The model exchange flavor provides only the simulation model: handles to inputs, outputs, continuous states are provided and simulation is done by external solvers. In co-simulation flavor, the FMU also contains the solver. This allows for exchanging simulation models with solvers dedicated to the physics of the model.

With Scade Hybrid, it is possible to import FMI 1.0 and 2.0 FMUs, in model-exchange or co-simulation kind:

- A model-exchange FMU defines ODEs, so it can be imported as an **hybrid** operator with the same inputs/outputs. It is mostly equivalent to an imported operator: an **hybrid** operator has the same behavior as the same operator compiled to an FMU;
- A co-simulation FMU is imported as a **node** operator. It can be activated periodically using an **activate** construct. The outputs of the FMU are piecewise constants (no interpolation is done);
- a co-simulation could be envisaged with SpaceEx, as it has been shown in [5].

As a result, an hybrid model can be made of not only continuous and discrete parts designed in Scade Hybrid, but also continuous simulation models designed in any FMI-compliant tool. This extends the simulation capabilities at model-level as specific models for the environment can be shared.

4.2.1 FMU wrapper generator

The goal of the FMU wrapper generator (`lib/fmu_generator.py`) is to:

[7] https://www.fmi-standard.org/

- Generate the files needed by the runtime, that is, `kcg_info.h` and *root*`_fmu.c` (which defines the array storing information about inputs/outputs, continuous states and zero-crossings);
- Generate the `modelDescription.xml` file describing the interface of the generated FMU.

These files are generated by reading the mapping file generated by KCG Hybrid and given as input of the script.

4.2.2 FMU import

The `lib/fmu_wrapper.py` script is used to import an FMU into a Scade Hybrid model. It generates a `.xscade` file defining the imported FMU as a Scade operator and `.c` and `.h` files that load the FMU DLL and call FMI functions. It supports FM1 1.0 and 2.0 co-simulation and model-exchange FMUs. To use this wrapper, one then needs to use the `--imp_fmu` option of the `pack_fmi.py` script.

Importing a model-exchange FMU A model-exchange FMU is imported as an hybrid operator with the same inputs/outputs:

```
hybrid fmu #pragma kcg nb_zeros Nz #end
#pragma kcg nb_cstates Nc #end N(...) returns (...);
```

Pragmas are added to specify the number of continuous states and zero-crossings used by this operator. These pragmas can be used for any imported hybrid operator. Continuous states (resp. zero-crossings) must be stored in an array called `kcg_cstates` (resp. `kcg_zeros`) stored in the context.

Importing a co-simulation FMU A co-simulation FMU is imported as a discrete node with a delayed causality:

```
node N(...) returns (...);
```

It means that the outputs at a given discrete step do not depend on the inputs at that date, but at the previous step. This is achieved by adding a delay (**pre**) on all the inputs of the node.

5 Conclusion

Mix of continuous and discrete behavior leads to several formalisms. Usually there is a need for a simulation master that drives the execution, that is to say when to call the solvers and when to activate the discrete part. The master could be the continous part or the discrete part. Regarding the continous part it can be modelled using causal or acausal semantics. Considerations on the final target (simulation or embedded controler) are also important.

Scade Hydrid is a formalism mixing continuous and discrete behavior and which is based on the well-defined semantics of the Scade language for discrete and deterministic behaviors. The continous part semantics is that of causal equations. All the properties of Scade (determinism, synchronous paradigm) are preserved and the interaction between discrete and continuous parts is well-defined using the notion of zero-crossing events. A few language constructs are introduced to deal with the notion of derivative and the events. This report provides the static and dynamic semantics of the Scade Hybrid language. With respect to existing research work, the Scade Hybrid, as a conservative extension of the Scade language, supports more contructs and data types.

A prototype of the Scade Hybrid code generator is under development. The prototype will check the semantics of the input model and will generate C code that will provide functions to compute the discrete part and the continous part. Regarding the continous part, the code will compute the current values of the zero-crossings, the derivatives and expected horizon for efficient time events implementations.

The generated code must be independent from any solver. It will rely on a back-end using a generic runtime which can be instantiated for a given solver. This mecanism has been experimented to generate FMI-compliant code in both model-exchange and co-simulation flavors. It as been also tested to generated standalone executable using the CVode solver.

The report describes the expected C interface and data types for the generated C code and gives the generic runtime specification. The presented work extends research works in different ways. First, the generated code will be C to fit customer environments. Second, the code generator will be an extension with minimal impact of the existing qualifed SCADE code C code generator. This approach paves the way for a certified code generator for Scade Hybrid as this strategy will concentrate the effort of certification on the new elements and will benefit of the already existing certification process[8].

This development provides a brick within the UnCoVerCPS toolchain. Indeed, it allows a smooth transition between the modelization of an abstract controller within its environment to

[8]This certification process is internal to Esterel Technologies, but available to certification authorities

its implemementation as an embedded certifiable software. Controller and environment state exploration done with SpaceEx is the mean to determine the exact conditions for the execution of the controller and which safety properties must be fulfilled by the final implementation. Scade Hybrid can be the bridge between the SpaceEx formalism and the Scade formalism used for actual implementation. Within Scade, the safety properties will be verified on the controller implementation. This proposed flow has been established through a fruitful collaboration within Esterel Technologies and Université Grenoble Alpes. During the next steps of the project, in collaboration with use-cases holders and in particular DLR, the proposed toolchain will be applied on examples.

A From Scade Hybrid to SpaceEx

Following the UnCoVerCPS proposal, a link between SpaceEx and SCADE will be established to connect verification and certified implementation of controllers. To establish that connection, it is necessary to analyze the semantics of both formalisms to understand in depth the similarities and differences. Furthermore, the project is an opportunity to introduce Scade Hybrid.

The rest of that section presents an analysis of the formalims based on a simple example and show the differences. Then, we propose a top-down workflow, where SpaceEx is the entry point to describe the system at an abstract level. Safety properties could be demonstrated at that level. The actual software implemenation can be done in Scade, and these properties must hold for the implementation.

This is on-going work. The next step is the finalization of the specification of the SpaceEx to Scade Hybrid translation and the realization of the translator.

A.1 Translation sketch on a small example

In this section, we will consider a simple Scade Hybrid example and try to create the corresponding SpaceEx model. This will illustrate the similarities between the two formalisms and a mapping between their concepts.

The example we consider is the `modes_ball` operator used in previous sections:

```
const y0:float64 = 2.0;
const g:float64 = 9.81;

hybrid simple_ball() returns (y:float64 last = y0; y_v:float64 last = 0.0)
let
    der y = y_v;
    activate if down y
    then y_v = - 0.8 * last 'y_v;
    else der y_v = - g;
    returns y_v;
tel
```

Translation principles The idea of the translation is to map each Scade equation to one base component in SpaceEx and to use a network component to link these base components. Since SpaceEx does not support hierarchical models, we have to consider the Scade Hybrid

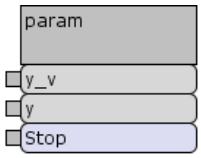
model where user-defined operators are replaced by their definition.

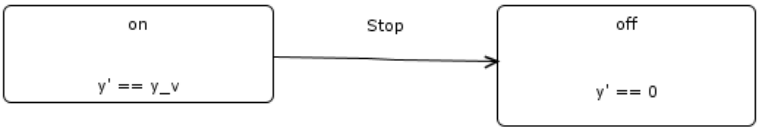
To deal with the fact that equations can be inactive (when they are contained inside an inactive automaton state), each equation is translated to a two-state automaton with an `on` state containing the behavior of the equation and an `off` state where nothing happens. Synchronized transitions are used to make sure that all equations in a given state are deactivated when leaving this state.

Translation of `simple_ball` Let's first consider the first equation of `simple_ball`:

```
der y = y_v;
```

It is mapped to the following SpaceEx model:



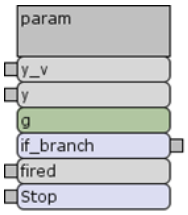


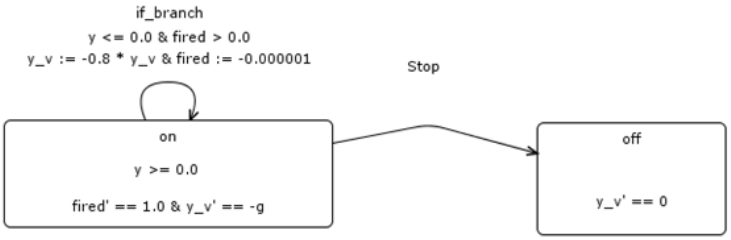
Each flow appearing in the equation, as input or output, appears as a parameter of the base component. The **der** equation is mapped to the definition of the derivative `y'`. The synchronized `Stop` transition leads to the `off` state, where the derivate is equal to zero, in order to hold the value of `y`.

The second equation is:

```
activate if down y
then y_v = - 0.8 * last 'y_v;
else der y_v = - g;
returns y_v;
```

It is mapped to:





The invariant in the body of the `on` state corresponds to the **der** equation in the **else** branch of the **activate**. The **then** branch, which resets the value of `y_v`, is mapped to the action on the transition, denoted `y_v := -0.8 * y_v`.

The event **down** `y` is active when `y` crosses zero from positive to negative. The **then** branch is activated only once at this instant. It is mapped to a guard `y <= 0` for the transition and to a constraint `y >= 0.0` inside the state, to make sure that the transition is taken as soon as possible.

The `fired` variable is here to encode the semantics of the Scade Hybrid discrete event in SpaceEx. In SpaceEx, transitions are non-deterministic: they can be fired at any time as long as the guard is true. In the example, since the transition resets the speed `y_v`, the guard of the transition `y <= 0.0` is still true. Without the additional constraints on `fired`, the transition could be fired multiple times when `y` reaches zero. The definition and constraints on `fired` guarantee that at least 0.000001 seconds must elapse between two successive firings of the transition `if_branch`.

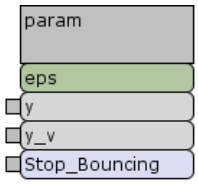
Translation of `modes_ball` The body of the operator is:

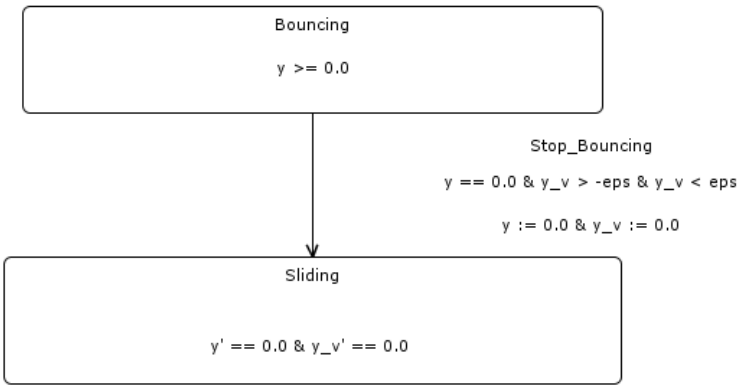
```
automaton
  initial state Bouncing
    y, y_v = ball();
  until if down y and y_v < eps restart Sliding;

  state Sliding
  let
    y = 0.0;
    y_v = 0.0;
```

```
    tel
returns y, y_v;
```

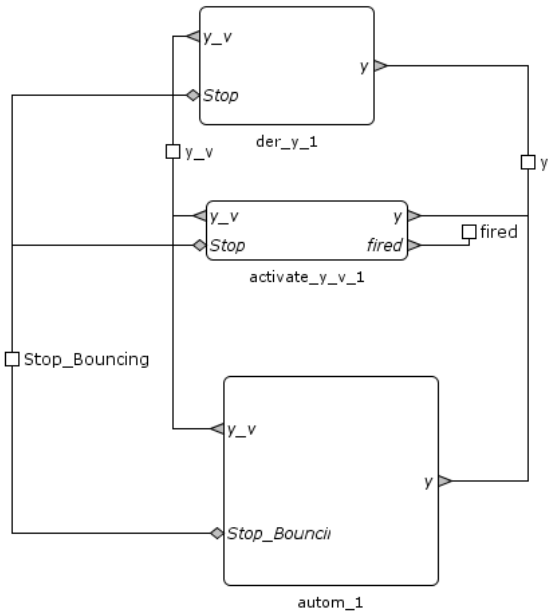
It is mapped to:



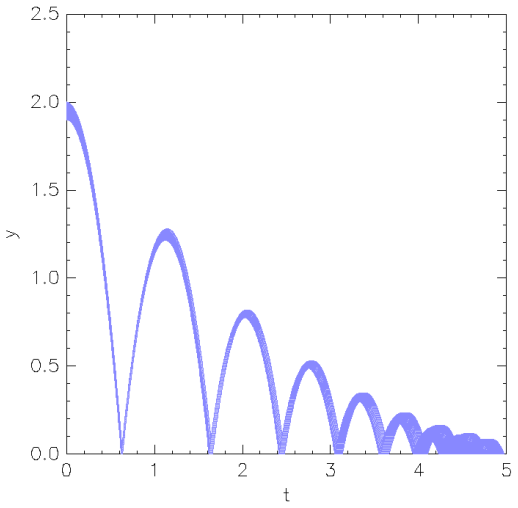


Each state of the Scade Hybrid is mapped to a SpaceEx location (we omit here the `off` location since this is the root operator which is always active). Note that the transition guard also includes `y_v > -eps`. Indeed, in the Scade Hybrid model, the check `y_v < eps` is done after the evaluation of the body of `ball`. It means that if **down** `y` is true, then `y_v` is always positive (because the **then** branch of `modes_ball` has been executed). A possible solution would be to force the `if_branch` transition to occur before `Stop_Bouncing`. In this example, it is simpler to just check that the absolute value of `y_v` is smaller than `y_v`, which is basically equivalent.

Putting it all together The following network component is used to link the base components described above:



This network components instantiates each base component and links the corresponding variables and transitions. Here is the result of the execution of the model in SpaceEx for `1.9 <= y <= 2.0`:



Issues raised by the example Most of the issues come from the fact that transitions are non-deterministic in SpaceEx, whereas they are deterministic in Scade Hybrid. It is sometimes possible to force a non-deterministic transition to become deterministic by putting the negation of the transition guard in the state staying condition. However, this is not possible as soon as the guard is a conjunction: its negation is a disjunction, which is not supported by SpaceEx. This is for instance the case in the mapping given for `modes_ball` above. It can also be an issue of the condition of the transition is still true after a discrete

event, as in the mapping of the **activate** of `simple_ball`. The next section will describe an extension of SpaceEx to model *urgent* transitions, which should help to bring the two formalisms closer.

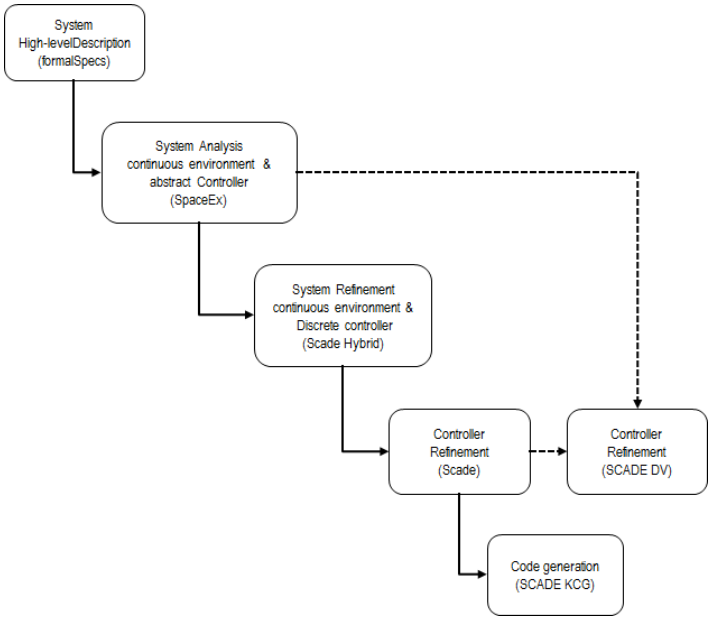
Another difference is that Scade Hybrid guarantees that weak transitions in an automaton are executed after the body of the state. This behavior must also be encoded in the SpaceEx model.

SpaceEx is also unable to model the discrete state of a Scade Hybrid model. Each possible state of the system must be mapped to a different location in SpaceEx model. This will result in a state explosion for any non-trivial discrete state, for instance when modeling a discrete controller.

A.2 Top-down workflow

Use of SpaceEx In this workflow, SpaceEx is used for system-level analysis. This analysis is done before or during the definition of the system high-level requirements. It is based on a model of the whole system and its environment. In particular, only an abstraction of the controller is used. *formalSpec* can be used to express the properties to be checked and translate them to hybrid automata.

The figure depicts the flow, where the black arrows denote the refined data passed between each steps, and the dashed arrows the properties to be verified.



The next paragraphs provide details on each steps.

Use of Scade Hybrid It is used for the design of low-level requirements. It is used to design and simulate of the controller satisfying the high-level requirements obtained from the

previous analyses and system requirements. Scade Hybrid allows to start from a continuous model of the controller and to refine it step by step into a discrete design. It is also possible in the same language to model the environment of the controller for simulation purposes.

Use of Scade Since Scade Hybrid is a superset of Scade, the controller designed in the previous phase can be directly given to Scade Suite KCG to generate automatically embeddable code for the controller. The rest of the process is the traditional one for Scade users.

XF: mettre diagramme cycle en V Scade Suite

Testing Another possible link is to extract from the analysis done on the SpaceEx model a contract that the controller implemented in Scade must respect. This contract can be the abstraction used in the SpaceEx model. It can be either verified by tests or proven formally using SCADE Design Verifier, the formal proof assistant of the SCADE Suite toolchain.

B Bibliography

[1] Albert Benveniste, Timothy Bourke, Benoit Caillaud, Bruno Pagano, and Marc Pouzet. A Type-Based Analysis of Causality Loops in Hybrid Modelers. In *17th International Conference on Hybrid Systems: Computation and Control (HSCC'14)*, page to appear, Berlin, Germany, April 2014. 15, 16

[2] Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet. A Hybrid Synchronous Language with Hierarchical Automata: Static Typing and Translation to Synchronous Code. In *ACM SIGPLAN/SIGBED Conference on Embedded Software (EMSOFT'11)*, Taipei, Taiwan, October 2011. 12, 16

[3] Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet. Divide and recycle: types and compilation for a hybrid synchronous language. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES'11)*, Chicago, USA, April 2011. 8, 16, 20

[4] Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet. Non-Standard Semantics of Hybrid Systems Modelers. *Journal of Computer and System Sciences (JCSS)*, 78(3):877–910, May 2012. Special issue in honor of Amir Pnueli. 16

[5] Sergiy Bogomolov, Marius Greitschus, Peter G. Jensen, Kim G. Larsen, Marius Mikucionis, Thomas Strump, and Stavros Tripakis. Co-simulation of hybrid systems with spaceex and uppaal. In *Proceedings of the 11th International Modelica Conference*, 2015. 27

[6] Timothy Bourke, Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet. A Synchronous-based Code Generator For Explicit Hybrid Systems Languages. In *International Conference on Compiler Construction (CC)*, LNCS, London, UK, April 11-18 2015. 8, 16

[7] Timothy Bourke and Marc Pouzet. Zélus, a Synchronous Language with ODEs. In *International Conference on Hybrid Systems: Computation and Control (HSCC 2013)*, Philadelphia, USA, April 8–11 2013. ACM. 16

[8] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proceedings of IFIP Congress*, volume 6, 1974. 7

[9] Thomas Moniot, Bruno Martin, and Jean-Louis Colaço. *The SCADE 6 language*, 2014. 7, 17