Scade 6: from a Kahn Semantics to a Kahn Implementation for Multicore

Jean-Louis Colaço
ANSYS SBU
Toulouse, France
Jean-Louis.Colaco@ansys.com

Bruno Pagano
ANSYS SBU
Toulouse, France
Bruno.Pagano@ansys.com

Cédric Pasteur
ANSYS SBU
Toulouse, France
Cedric.Pasteur@ansys.com

Marc Pouzet
École normale supérieure
Paris, France
Marc.Pouzet@ens.fr

***Abstract*—SCADE is an environment for developing critical embedded software that is used for more than twenty years in various application domains like avionics, nuclear plants, transportation, automotive. It comes with a language and a code generator which complies with the highest safety standards like DO-178C, IEC 61508, EN 50128, IEC 60880 and ISO 26262.**

The language has been founded on the pioneering work by Caspi and Halbwachs on Lustre. In 2008, a major revision of the language and compiler, named 'Scade 6', was released. One of its novelty was a smooth integration of the traditional data-flow style of Lustre with control-structures inspired from those of Esterel and SyncCharts, with static/dynamic semantics and a compilation inspired from Lucid Synchrone. In particular, it relies on four dedicated type systems — typing, clock calculus, causality analysis, initialization analysis — and a compilation through source-to-source transformations into a minimal clocked data-flow language, based on a Kahn semantics, that is translated to imperative code.

One ongoing work is the generation of code for multi-core architectures. Because of the intrinsic deterministic parallelism of Scade, we propose a solution that relies on annotations that specify what must be executed concurrently but do not change the semantics.

The paper is a survey of past to ongoing work on Scade 6 language definition and implementation.

***Index Terms*—synchronous languages, compiler, multi-core.**

I. Introduction

Synchronous languages were introduced about thirty years ago by the works on three academic languages: SIGNAL [1], ESTEREL [2] and LUSTRE [3]. These *domain-specific languages* were targeted at real-time control software, allowing users to write modular and mathematically precise system specifications, to simulate, test and verify them, and to automatically translate them into embedded code.

These languages were all founded on the *synchronous approach* [4] where a system is modeled ideally, with communications and computations supposed to be instantaneous, with (1) *a priori* guaranteed important safety properties (determinism, deadlock freedom, bounded execution time and space) and (2) an *a posteriori* verification that the generated implementation is fast enough.

These foundations immediately raised interest in industries having to deal with safety-critical applications implemented in software or hardware, and in particular, those assessed by independent authorities and following certification standards [5].

This is the context in which SCADE[1] was initiated in the mid nineties, with the support of two companies, Airbus and Merlin Gerin, and in collaboration with the research laboratory VERIMAG in Grenoble, and the software publisher VERILOG [6]. Since 2000, SCADE is developed by ANSYS/ESTEREL-TECHNOLOGIES.[2]

SCADE is an integrated development environment (IDE) with a graphical block diagram editor and tool support to represent synchronous programs. In the first versions, the underlying language of the SCADE tool was essentially LUSTRE V3 [7], augmented with a few specific features requested by users but minor in terms of expressiveness. This situation persisted until version 5 of SCADE. To support the development of critical applications without having to verify the consistency between the model and the generated code, a 'qualified code generator' called KCG was developed. Its first version was released in 1999 and has been used, since then, in software projects with the most demanding safety levels of many standards (DO-178C, IEC 61508, EN 50128, IEC 60880 and ISO 26262), where high confidence in automation is expected. It is unique in the field of safety critical embedded software and contributed to the industrial success of SCADE. It also demonstrates the interest of a semantically well-defined language in the context of qualification processes.

In 2008, a new language named 'SCADE 6' and its compiler were released [3]. The objective was to widen the scope of applications developed with SCADE, yet keeping the ability to certify its code generator with the highest standards. SCADE 6 introduced several new language features, like the mix of data-flow equations and hierarchical state machines, new compile-time checks expressed by four different dedicated type systems, and a compilation through a sequence of source-to-source transformations into a minimal *clocked data-flow language*. The language features and its design are described in [8]. The qualified code generator (KCG) for this new version of the language was developed from scratch using state-of-the-art techniques in language design and implementation.

[1] SCADE stands for *Safety-Critical Development Environment*

[2] http://www.ansys.com/products/embedded-software/ansys-scade-suite

[3] To distinguish between the environment and its underlying programming language, we write SCADE for the former and SCADE (with small capitals) for the later.

On the embedded target side, the use of multi-core architectures is now considered for safety critical systems and there is a pressing demand for a SCADE compiler that targets those architectures. The implicit (deterministic) parallelism of synchronous block diagrams make them good candidates for a parallel implementation. This is not at all a new observation: several works have addressed the automatic distribution of synchronous programs [9], the implementation of synchronous models on a multi-task OS either running on a single core [10], [11] or multi-core [12] to cite a few.

In this paper, we informally go through the main design decisions of SCADE 6 and show how they were used in the development of KCG. We then report on an approach we have followed for targeting a multi-core platform. Its principle is to rely on annotations on the model that no not affect the semantics but tells the compiler to generate independent tasks that communicate through channels. The generated set of tasks form a Kahn process network (KPN) [13] which computes the very same streams as the source model. The actual implementation of this set of tasks on the final platform, as well as its timing analysis, is done afterwards and outside of the language.

The paper is organized in the following way. Section II reminds the Kahn semantics of the core data-flow kernel of SCADE 6 and the purpose of the type-based static checks. Section III presents the qualified compiler KCG. Section IV presents the way we propose to address multi-core programming in SCADE. We conclude in Section V.

II. SCADE 6 INTUITIVE SEMANTICS

The main extension introduced in SCADE 6 compared to LUSTRE is the mix of dataflow equations and hierarchical state-machines [14]. Its design goes further with the definition of all the static and sufficient conditions for a program to be correct, conditions that are expressed as type systems, in the style of LUCID SYNCHRONE [15]. Code generation is done when all static conditions are fulfilled. The language is built on top of a small dataflow language kernel, reminiscent of LUSTRE. High level constructs are progressively rewritten into this language kernel which is then compiled into sequential code.

The type systems have been formalized for the whole SCADE 6 language but their correction has been proved only on the dataflow kernel. These proofs can be found in the related papers quoted below.

In this section we remind the reader of the stream semantics of the dataflow kernel of SCADE 6, we give some insights of the way we defined the state machine semantics, and then go through its four type systems.

A. A dataflow kernel

The kernel language used in SCADE 6 is essentially that of [16]. It is a variant of LUSTRE, with the operator **current** replaced by **merge** and extended with a reset construct. Here, we consider a simpler version to remind its semantics over streams.

$$\begin{array}{rcl} e & ::= & i \mid x \mid op(e,...,e) \mid f(e,...,e) \\ & & \mid \textbf{pre}(e) \mid e \text{ -> } e \\ & & \mid e \textbf{ when } e \mid \textbf{merge}(e,e,e) \end{array}$$

The language of expressions (e) is made of constants (i) like integers or Booleans, identifiers (x), imported operator (op) (e.g., arithmetic operators like $+$, $-$) applied point-wise to streams, the application of an operator (f), an uninitialized unit delay (**pre**) and a stream initialization operator (->), the filtering of a stream according to a Boolean stream (**when**) or the combination of two streams according to a boolean condition (**merge**). Note that these Boolean conditions appearing in **when** and **merge** are called clocks.

A system is defined by an operator (denominated a 'node' in LUSTRE and SCADE), that is, a function that transforms a set of input streams into a set of output streams. The body of the function is made of a set of equations of the form $x = e$ where defined variables are either outputs of the function or local to the function. E.g., the following definition computes the cumulative function of input `u`, that is, its n-th value is the number of true values of input `u` up to instant n. Expression `0 -> pre o` is a unit delay `o` whose first output is `0` and its n-th value is the $n-1$-th value of `o`. The conditional **if**/**then**/**else** and `+` apply point-wise.

```
node counting_events(u:bool)
  returns (o: bool)
var o: int;
let
  v = if u then 1 else 0;
  o = v + (0 -> pre o);
tel;
```

There are several useful semantics for LUSTRE and its variants. The simplest one is based on the semantics for dataflow networks given by Kahn in his seminal paper [13] that we remind below. Let T be a set, nil a particular value of this set. Let T^n be the set of sequences of length n made by concatenating elements from T. ϵ is the empty sequence. $T^\star = \cup_{n=0}^{\infty} T^n$ is the Kleene star operation. We write $v.s$ for a sequence whose first element is v and tail is s. T^∞ is the set of finite and infinite sequences of type T, that is, $T^\infty = T^* \cup T^\omega$. The set $(T^\infty, \leq, \epsilon)$, with $\leq$ the prefix order between sequences, ϵ the minimum element, is a complete partial order (cpo). Then the Kleene theorem applies: if $f : T^\infty \rightarrow T^\infty$ is a continuous function, an equation $x = f(x)$ defines the sequence $x^\infty = lim_{n\rightarrow\infty}(f^n(\epsilon))$ which is the smallest fixpoint of f. This construction generalizes to the case of mutually recursive equations. Hence, a function from sequences to sequences whose outputs are defined by a set of mutually recursive equations over sequences is also continuous, provided all the functions it calls are continuous.

An expression e with (free) variables $x_1, ..., x_n$ that are sequences is interpreted as a continuous function of these variables [17]. For that, we define the semantics of primitives in Figure 1. They are all continuous functions. A constant i is

$$
\begin{array}{lcl}
lift^0(v) & = & v.lift^0(v) \\
lift^1(op)(v.s) & = & op(v).lift^1(op)(s) \\
lift^1(op)(\epsilon) & = & \epsilon \\
lift^2(op)(v_1.s_1, v_2.s_2) & = & op(v_1, v_2).lift^2(op)(s_1, s_2) \\
lift^2(op)(s_1, s_2) & = & \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
(v_1.s_1) \rightarrow (v_2.s_2) & = & v_1.s_2 \\
(s_1) \rightarrow (s_2) & = & \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
pre(s) & = & nil.s \\
when(v.s, true.c) & = & v.when(s, c) \\
when(v.s, false.c) & = & when(s, c) \\
when(s_1, s_2) & = & \epsilon \text{ if } s_1 = \epsilon \text{ or } s_2 = \epsilon \\
merge(true.c, v.s_1, s_2) & = & v.merge(c, s_1, s_2) \\
merge(false.c, s_1, v.s_2) & = & v.merge(c, s_1, s_2) \\
merge(true.c, \epsilon, s_2) & = & \epsilon \\
merge(false.c, s_1, \epsilon) & = & \epsilon \\
merge(\epsilon, s_1, s_2) & = & \epsilon
\end{array}
$$

Fig. 1. The stream interpretation for primitives

interpreted as an infinite sequence made of that constant, that we write $lift^0(i)$. Function $lift^1(.)(.)$ defines the semantics of the pointwise application of a unary operator op; $lift^2(.)(.)$ defines the case for a binary operator. Function $pre(.)$ defines the semantics of the un-initialized unit delays. Function $(.) \rightarrow (.)$ defines the semantics of the stream initialization. Function $when(.,.)$ defines the semantics of the filtering operator **when**; $merge(.,.,.)$ defines the semantics of the operator **merge**.

The interest of this semantics is its simplicity: a system is a function over sequences, hiding implementation details. However it is not well-adapted to express operational properties like the existence of an execution with bounded memory, synchronization issues and the actual size of a buffer and the ability to produce statically scheduled code. [4] To address those questions, the Kahn semantics can be turned into a synchronous Kahn semantics that use explicit present/absent values [18]–[20].

B. The hierarchical state machines of SCADE 6

In the language kernel we have considered, the body of a function is defined by a set of equations. In SCADE 6, those equations can be arbitrarily composed with state machines. A state machine is a special form of equation: it is defined by a set of states, each of them containing a set of equations (possibly containing state machines themselves) and conditions to go from an active state to an other one. The variables defined in one state can either be local to the state or global to the automaton. The full language preserves the static single assignment (SSA) property of the language kernel: only one state is active per synchronous cycle. If a variable is defined in two different states, only one equation defines its value for a given cycle. Moreover, an automaton introduces a set of scopes, one per state allowing to define variables that only exist when this state is active. [5]

The semantics of state machines is defined by their translation to the dataflow kernel. This is done by capturing the structure of scopes with clocks based on an enumeration: one enumerated value per state and one enumerated type per state machine. The exclusivity between states is ensured by a simple encoding: the stream that defines the current active state has a single value at every reaction. Its current value depends on the previous active state and the transitions in the current state.

The **merge** operator generalizes to n streams on n clocks derived from the same enumeration. Once these items are introduced, each local variable declared in a scope becomes a variable (with the appropriate renaming to avoid collisions) declared at the top level of the encompassing operator with the clock introduced for its scope. Then all the right-hand sides of the different exclusive definitions are on exclusive clocks and can be merged. A bottom up application of this principle allows to translate arbitrary nested scopes.

These translation principles shows that scopes and their activation are, in SCADE 6, just an alternative way to manipulate clocks. This specification is given in [14] with all the necessary details for an implementation.

C. Static semantics

The static semantics encompasses the invariants that a program must satisfy before considering its execution. For SCADE 6 we express them as typing problems so that, quoting Robin Milner, "well-typed programs cannot go wrong" [21]. Without going into the details of the types, we give for each of them what "going wrong" means. The type systems are applied in the order they are presented below.

1) Type checking: This analysis checks types, in the usual sense. For SCADE 6, the following design choices have been taken:

- The type of variables must be declared; a type is either an enumerated set of values, a record, an array parameterized by a size expression whose value has to be known at compile time, or an abstract type.
- Type equivalence is based on structural equality.
- The language provides a number of built-in type classes, like **numeric** and **integer**. E.g., **int8**, **int16**, **int32**, etc., are elements of the class **integer**.
- Types can be polymorphic and possibly constrained by the type classes **numeric**, **float**, **integer**, **signed**, or **unsigned**.
- Functions may be parameterized by a size. This parameter can be used in an array type, for example.

Well-typed programs satisfy the following properties:

- *function arguments have the expected type so that the program does not have type errors at run-time;*
- *array accesses are within array bounds.*

[4] Using length arguments, it is enough to justify the so-called clock calculus of LUSTRE to ensure that the composition of two bounded memory networks is a bounded one [17].

[5] Remember that only the body of a node introduce a scope in LUSTRE.

2) Clock checking: The clock analysis ensures that programs can be executed synchronously, that is, the corresponding Kahn process network does not need any, possibly unbounded, buffering mechanism. Said differently, the implementation of a stream of values of type t only needs to store the current value. As SCADE 6 forbids recursion (e.g., the Kahn process network defining the sieve of Eratosthene [13] is forbidden), then a well-clocked program defines a network that can be implemented with a bounded memory, provided that functions (imported or not) applied point-wise to a stream also run in bounded memory.

This property can be formulated as:

- *a well-clocked program can be implemented in bounded memory, provided that all the imported functions it uses do so.*

This property is fundamental for safety-critical embedded applications.

3) Causality analysis: The purpose is to ensure that the synchronous Kahn process network does not deadlock. SCADE 6 adopts the same simple policy as LUSTRE: all cycles defined by the read/write dependences between variables must cross a unit delay (**pre**) [22]. This dependence relation is syntactical, in the sense that it does not depend on actual values of signals or their clock, contrary to ESTEREL [23] and SIGNAL [24], for example. The main property the analysis ensures is:

- *The streams in a causally correct* SCADE 6 *program are such that the definition of the value of a stream neither depends on itself nor on values that appear after in this stream.*

The causality implemented by the compiler is actually a bit stronger and is such that the stream definitions can be evaluated in a statically chosen order. This leads to the corollary that:

- *A causally correct program can be compiled into statically scheduled sequential code that runs in bounded time provided that all imported functions do so.*

As for clock checking, the causality analysis guarantees the existence of an upper bound on the time necessary to compute a reaction also known as Worst Case Execution Time (WCET).

4) Initialization analysis: In LUSTRE, the unit delay operator **pre** returns a sequence that is *un-initialized* at the first instant. The un-initialized delay is used in combination with the initialization operator `->` which allows to define the first value for a stream. A typical use being in an equation:

```
x = x0 -> f(pre x)
```

which defines the stream `x` whose first value is the first value of `x0` and following values (non-first ones) are equal to the corresponding values of `f(pre x)` i.e. all the values but the first.

The un-initialized delay is thus a source of non determinism in the program since the actual implementation for a nil is not defined by the language. The purpose of the initialization analysis is to ensure that the outputs of the main operator[6] never depend on those values. The analysis is also expressed as a dedicated type system, applied modularily to every function definition and computing a type signature for each of them.

- *A main* SCADE 6 *program that has passed the initialization analysis is deterministic, i.e., applied to two equal sequences of inputs, it returns two equal sequences of outputs*

This property focuses on the main operator outputs because this is where determinism matters; the property is weaker for internal flows to allow the use of operator (like **pre**) able to produce a nil. Determinism is an important property for critical systems, certification standards require in general to provide evidence of the determinism of the software. The KCG compiler of SCADE guarantees this property.

III. SCADE 6 COMPILER

The SCADE 6 code generator is qualified for the main standards of safety critical industry (DO-178C, IEC 61508, EN 50128, IEC 60880 and ISO 26262). This means that it can be used in the software development of a safety critical system without having to verify that the code produced actually implements the SCADE model. The present section gives a few facts about this code generator and its development as a qualified tool.

A. Compiler Organization

The organization of the compiler (KCG) is rather classic. Static analyses are applied in sequence right after parsing, in the order they have been presented in section II-C. If they all succeed, code generation starts with a sequence of source-to-source transformations that progressively rewrite high-level constructs (e.g., hierarchical state machines, activation conditions) into the clocked data-flow kernel. Then, the data-flow kernel is translated into an intermediate sequential language. Finally, target imperative code (C or Ada) is emitted. Figure 2 summarizes these steps at a high level; bibliographic references are given on the arrows.

Within the transformations, many optimizations are performed on the data-flow form (dead-code elimination, constant propagation, common sub-expression elimination, etc.). The scheduling in the data-flow compilation implements heuristics aiming at minimizing memory size. Control structures are merged in the sequential representation.

B. Qualified Development

Qualification is based on traceability between a specification and implementation. The source and intermediate languages have been formally specified together with the static semantics (defined by inference rules) and source-to-source transformations (defined by rewrite rules). Those specifications are used by the development team to implement the compiler and by an independent verification team to test it.

[6]The main of a SCADE application is also called *root node*.

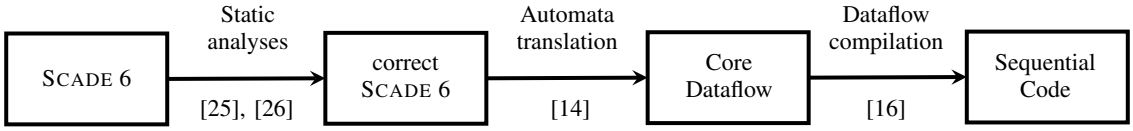

Fig. 2. SCADE 6 Compiler Organization.

For the implementation, we chose OCAML [27] which in 2005 presented quite a challenge for a qualified tool. Indeed, certification standards often push companies to use well established technologies. We thus had to provide convincing evidence that OCAML was well adapted to write a compiler. The argumentation was built on the small distance between the formal specification and its implementation in OCAML. This industrial use of OCAML in a certified context is detailed in [28] and [29].

The current version of SCADE KCG comprises approximately fifty thousands lines of code (50 KLoC) and uses a simplified OCAML runtime to satisfy the objectives of the standards. The formalized static semantics for the whole input language is about one hundred pages long and has been updated over more than ten years to integrate new language features. The detailed design is more than one thousand pages long.

IV. CODE GENERATION FOR MULTI-CORE TARGETS

Multi-core targets are starting to be considered for safety-critical applications because of their better power efficiency, the limited availability of single-core targets and the need for more computing power.

This section describes a flow to generate a code that can be integrated on a multi-core target. We only give here an overview of the flow adopting a SCADE user point of view. The details of this code generation will be the subject of a future communication.

As a first step, we have decided not to extend the language with dedicated constructs and just provide some compiler pragmas to exploit the intrinsic parallelism of SCADE programs. The main advantage of this approach is that it applies on legacy SCADE 6 models. On the target side, we generate C code and make no hypothesis on the architecture. Our goal is to allow SCADE users to take their model and generate a code structured to exploit computational resources of their target.

A. Overview

The objective of this ongoing work is to:

- generate code that can be executed efficiently on multi-core or many-core targets;
- preserve the deterministic semantics of the language;
- be target-independent since the hardware and software architectures used in critical embedded systems differ widely from one domain to another (with or without OS, with or without shared memory, number of cores, ...).

Note that this work focuses on the structure of the generated code to satisfy these objectives, in the vein of the work on automatic distribution of synchronous programs [9]. We are not looking at performance issues (like scheduling or efficient implementation of communications). We are also not focusing on memory interferences [30], which can impact the ability to compute the *Worst Case Execution Time* (WCET) of the code, as they are specific to a platform. Our goal is to provide a flow to make sure that these issues can be dealt with at integration level, with no impact on the functional model. This is important for industrial use, where, in general, the team in charge of the design is different than the team in charge of the integration on the target platform.

The proposed flow is the following:

1) The user designs the SCADE model or reuses an existing model.
2) The user annotates the model to express potential parallelism. An operator instance is annotated with a dedicated pragma to indicate that it should be extracted into a task. The call to this operator will be replaced with channel operations, so that this instance can be executed separately, eg. in another thread. Note that this annotation does not change the semantics of the model, only the shape of the generated code.
3) The Multicore Code Generator (MCG) generates C code for each task as well as traceability information mapping the C code to the input model and a complete description of the generated KPN.
4) A target-specific integration script generates code (based on the generated KPN) to allocate and schedule the generated tasks to threads/cores and to implement communications.

Steps 2 to 4 are done during integration.

Tasks can be extracted from any instance in the model. For instance, it is possible to extract instances inside an activate block or a state machine. Tasks are not restricted to instances in the root operator. Tasks can also be nested, meaning that a task can be extracted from an instance inside an operator which is already itself a task.

This flow ensures a clear separation of concerns between the different activities involved in running the code on a multi-core target. In particular, the functional architecture of the model can remain independent of target integration matters, in the sense that its organization does not need to know about the final code mapping to available cores.

The model focuses on what is the function to implement,

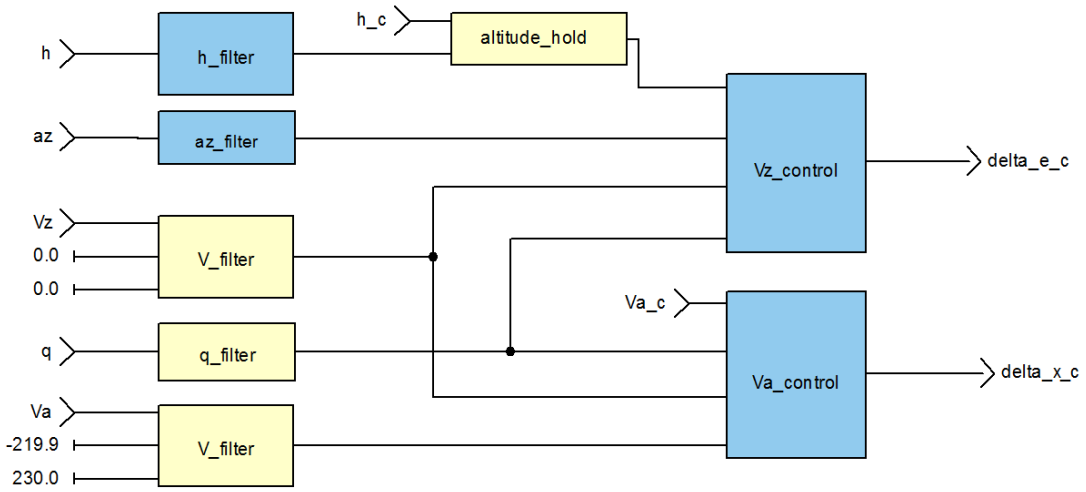

Fig. 3. Rosace in SCADE

not where the necessary computation takes place. The code generator takes care of splitting parts of the model into tasks ready for integration on the target.

B. Generated code

MCG generates tasks that communicate via one-to-one channels, that is, a Kahn process network [13]. One task executes the root operator of the SCADE model. One task is generated for each operator instance annotated in the input model. This task receives data on an input channel, calls the operator and then sends the result on an output channel.

Tasks are similar to the objects described in [16]. Each task has a context, containing its memories and several methods:

- a *reset* method to initialize the memories;
- one or several *cycle* methods.

The main difference with KCG generated code and [16] is that the cycle method can be split into several methods. The objective of this splitting is to reject communication out of the sequential code and to not over-constrain the possible schedules. A contribution of MCG is to preserve the possible schedules in order not to interfere with the efficiency of the multi-core integration. This problem is similar to the issue of modular compilation discussed in LUSTRE [31]. The methods of the same task can communicate directly through the context. Each cycle method has input and output channels to communicate with other tasks. The body of each method contains sequential code, generated as usual from the SCADE model.

Figure 3 shows a SCADE version of the ROSACE case study [12]. Annotated instances, that is, specifying that an independent task must be generated, are colored in blue. The corresponding generated KPN is depicted in Figure 4. One task is created for each annotated instance. The root task is made of three methods (root_1, root_2 and root_3). The channels to implement are identified (c1 to c9) with their source and destination. Figure 5 shows a possible schedule for the thread implementing the root node; the communications (green boxes) are part of the integration code to be written by the user (may be automated by platform specific scripts).

The KPN model, or more generally the usage of isolated tasks communicating through message passing, is pretty common for multi-core execution, especially for safety-critical system. It guarantees isolation between tasks, which only communicate through channels. Communication points between cores are also well-identified and separated from regular computations. This allows for more control over memory interferences.

MCG itself does not provide a solution to mitigate memory interferences. It does however make sure that it remains an integration problem, with no impact on the functional model. Properties such as isolation are guaranteed provided that the integration code respects the usage conditions of MCG generated code.

C. Target integration

The target integration consists in allocating the tasks generated by MCG to threads/cores and to implement communications. The KPN is described in the traceability information generated by MCG. [32] describes an implementation for the PXROS operating system on Infineon Aurix platform. It is also straightforward to implement communications using traditional data structures like semaphores or C11 atomics.

The KPN model gives a formal model of the code generated by MCG. It is used to specify the usage conditions of this code, for instance how communications should be implemented or the dependencies between tasks. The target integration must ensure that these conditions are verified, to ensure that the behavior of the code is the same as the sequential code. For instance, we guarantee that the generated KPN only needs buffers of size one. We can also rely on the literature for scheduling and allocation algorithms for KPNs [33].

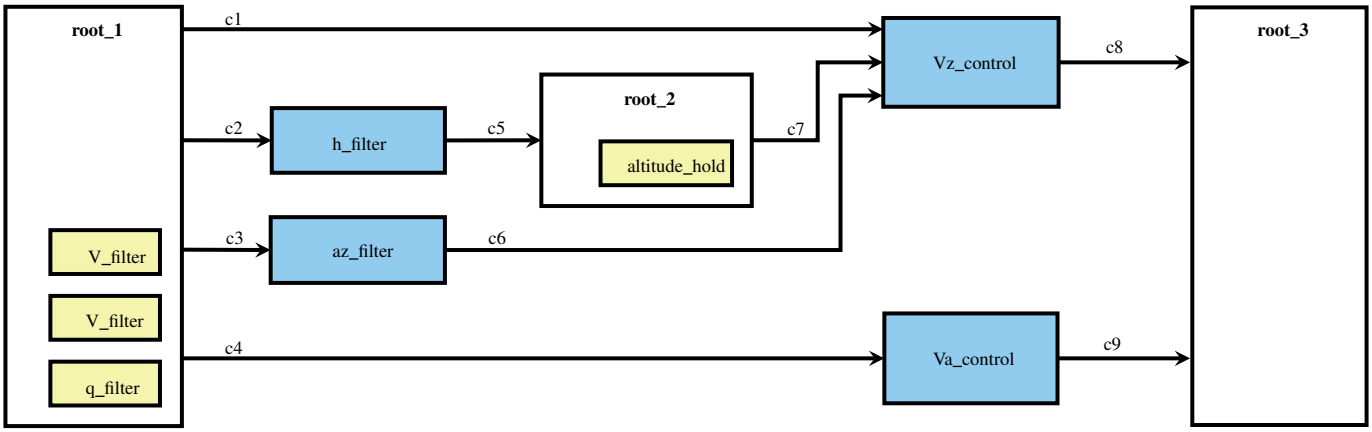

Fig. 4. Rosace on target

D. Impact on the code generator

As we said above, the language considered by MCG and its semantics is SCADE 6, thus all the front-end is shared with KCG. The only additional checks are relative to the added pragmas and their compatibility with other compiling directives like operator inlining.

The communication channels are then introduced in the core dataflow form. The splitting of the operator bodies is done in a new intermediate language that allows several cycle methods per operator; this is a kind of declarative version of the imperative *simple object-based language* introduced in [16]. The algorithm that implements this step is inspired by the modular compilation of LUSTRE proposed in [31].

The sequential parts of the code i.e. the body of the different methods that implement an operator are scheduled and generated the same way KCG does.

E. An extension: pipelining

In some cases, it may not be possible to parallelize an existing SCADE model because of the sequentiality of the computations. A classic solution in that case is to use pipelining and to add memories between the different parts of the computation. It takes more steps to compute the result, but the different parts of the pipeline can be executed in parallel.

Pipelining can be achieved in the same way in SCADE, by adding a unit delay (`fby`), between different operators. This results in a channel with an initial token, containing the initial value of the `fby` operator. It is interesting to note that for this extension the KPN model once again allows to describe the generated code using existing concepts and algorithms.

V. CONCLUSION

This paper illustrates the relevance of Kahn process networks to address parallelism; it is used here for both language design and as an execution model used to abstract multi-core architectures. These two parallelisms correspond to different phases of a system design:

- its functional definition with SCADE 6 and
- its implementation on a multi-core architecture with the help of MCG.

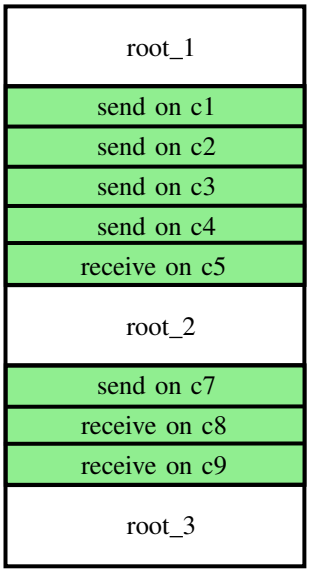

Fig. 5. A schedule for the Rosace root

When designing a function it is good to concentrate on what has to be computed regardless on when it has to be scheduled; that is what synchronous languages allow. And when implementing on a multi-core it is not necessary to know what the pieces of code to integrate are computing but only timing aspects (worst-case execution time, worst-case communication time) to make the placement choices.

The originality of the proposition is to not correlate these two steps and allow to have a functional architecture that differs from the implementation one.

Another interesting point is the preservation of the semantics. The language itself is not changed to adapt to multi-core targets; it is still the same and thus deterministic with all the appreciated properties when it comes to verification or

maintenance. The annotations introduced to identify the tasks are just compiler directives that affect the organization of the generated code, not the behavior of the model.

This separation is also interesting from an industrial point of view because it allows to have different implementation strategies depending on the actual hardware. It often happens that a given embedded function is implemented on different micro-controllers with different number of cores and different computation power per core. In automotive for instance, top-end cars can have more expensive hardware and share some functions with lower-end ones.

MCG will be part of the next release of SCADE. It is not qualified yet and the use of multi-core on the most critical systems is still a work in progress for both industry and certification authorities. This work is a contribution to this reflection. It provides a partial answer to the questions raised by these architectures when used in a safety critical context.

REFERENCES

[1] A. Benveniste, P. LeGuernic, and C. Jacquemot, "Synchronous programming with events and relations: the SIGNAL language and its semantics," *Science of Computer Programming*, vol. 16, pp. 103–149, 1991.

[2] G. Berry and G. Gonthier, "The Esterel synchronous programming language, design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.

[3] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous dataflow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, September 1991.

[4] G. Berry, "Real time programming: Special purpose or general purpose languages," *Information Processing*, vol. 89, pp. 11–17, 1989.

[5] Berry, G., "Formally unifying modeling and design for embedded systems - a personal view," in *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications: 7th International Symposium, ISoLA 2016*, T. Margaria and B. Steffen, Eds. Corfu, Greece: Springer International Publishing, October 10-14 2016, pp. 134–149, proceedings, Part II.

[6] N. Halbwachs, "A synchronous language at work: the story of Lustre," in *Third ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, Verona, Italy, July 2005.

[7] N. Halbwachs and P. Raymond, "A tutorial of Lustre," 2002, http://www-verimag.imag.fr/Publications-Synchrones.html.

[8] J.-L. Colaço, B. Pagano, and M. Pouzet, "Scade 6: A Formal Language for Embedded Critical Software Development," in *International Symposium on Theoretical Aspects of Software Engineering (TASE 2017)*, Sophia Antipolis, September 2005.

[9] A. Girault, "A survey of automatic distribution method for synchronous programs," in *International Workshop on Synchronous Languages, Applications and Programs (SLAP)*. Edinburg, UK: ENTCS, April 2005.

[10] P. Caspi, N. Scaife, C. Sofronis, and S. Tripakis, "Semantics-preserving multitask implementation of synchronous programs," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 2, pp. 1–40, 2008.

[11] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens, "Multi-task implementation of multi-periodic synchronous programs," *Discrete Event Dynamic Systems*, vol. 21, no. 3, pp. 307–338, 2011.

[12] C. Pagetti, D. Saussié, R. Gratia, E. Noulard, and P. Siron, "The rosace case study: From simulink specification to multi/many-core execution," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*. IEEE, 2014, pp. 309–318.

[13] G. Kahn, "The semantics of a simple language for parallel programming," in *IFIP 74 Congress*. North Holland, Amsterdam, 1974.

[14] J.-L. Colaço, B. Pagano, and M. Pouzet, "A Conservative Extension of Synchronous Data-flow with State Machines," in *ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey city, New Jersey, USA, September 2005.

[15] M. Pouzet, *Lucid Synchrone, version 3. Tutorial and reference manual*, Université Paris-Sud, LRI, April 2006.

[16] D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet, "Clock-directed Modular Code Generation of Synchronous Data-flow Languages," in *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tucson, Arizona, June 2008.

[17] P. Caspi, "Clocks in dataflow languages," *Theoretical Computer Science*, vol. 94, pp. 125–140, 1992.

[18] P. Caspi and M. Pouzet, "Synchronous Kahn Networks," in *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Philadelphia, Pensylvania, May 1996.

[19] ——, "A Co-iterative Characterization of Synchronous Stream Functions," in *Coalgebraic Methods in Computer Science (CMCS'98)*, ser. Electronic Notes in Theoretical Computer Science, March 1998, extended version available as a VERIMAG tech. report no. 97–07 at `www.di.ens.fr/~pouzet/bib/bib.html`.

[20] S. Boulmé and G. Hamon, "Certifying Synchrony for Free," in *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, vol. 2250. La Havana, Cuba: Lecture Notes in Artificial Intelligence, Springer-Verlag, December 2001.

[21] R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and System Science*, vol. 17, pp. 348–375, 1978.

[22] N. Halbwachs, P. Raymond, and C. Ratel, "Generating efficient code from data-flow programs," in *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.

[23] G. Berry, "The constructive semantics of pure esterel," 2002, draft book. Available at: http://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.pdf.

[24] T. Amagbegnon, L. Besnard, and P. L. Guernic., "Implementation of the data-flow synchronous language signal," in *Programming Languages Design and Implementation (PLDI)*. ACM, 1995, pp. 163–173.

[25] J.-L. Colaço and M. Pouzet, "Type-based Initialization Analysis of a Synchronous Data-flow Language," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 6, no. 3, pp. 245–255, August 2004.

[26] J.-L. Colaço and M. Pouzet, "Clocks as First Class Abstract Types," in *Third International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, Pennsylvania, USA, october 2003.

[27] X. Leroy, "The Objective Caml system release 4.03. Documentation and user's manual," INRIA, Tech. Rep., 2017.

[28] B. Pagano, O. Andrieu, B. Canou, E. Chailloux, J.-L. Colaço, T. Moniot, and P. Wang., "Certified development tools implementation in objective caml," in *International Symposium on Practical Aspects of Declarative Languages (PADL)*, ser. Lecture Notes in Computer Science. Springer-Verlag, January 2008.

[29] B. Pagano, O. Andrieu, B. Canou, E. Chailloux, J.-L. Colaço, T. Moniot, P. Wang., and P. Manoury, "Experience report: Using objective caml to develop safety-critical embedded tools in a certification framework," in *International Conference on Functional Programming (ICFP)*. ACM, September 2009.

[30] P. Bieber, F. Boniol, Y. Bouchebaba, J. Brunel, C. Pagetti, O. Poitou, T. Polacsek, L. Santinelli, and N. Sensfelder, "A model-based certification approach for multi/many-core embedded systems," in *Embedded Real-Time Software and Systems (ERTS'18)*, 2018.

[31] M. Pouzet and P. Raymond, "Modular Static Scheduling of Synchronous Data-flow Networks: An efficient symbolic representation," in *ACM International Conference on Embedded Software (EMSOFT'09)*, Grenoble, France, October 2009.

[32] B. Pagano, C. Pasteur, G. Siegel, and R. Knížek, "A model based safety critical flow for the AURIX multi-core platform," in *Embedded Real-Time Software and Systems (ERTS'18)*, 2018.

[33] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computing Surveys (CSUR)*, vol. 31, no. 4, pp. 406–471, 1999.