SCADE: Synchronous design and validation of embedded control software

Gérard Berry

Esterel Technologies
gerard.berry@esterel-technologies.com
www.esterel-technologies.com

Abstract. We describe the SCADE synchronous approach to model-based embedded software design, validation, and implementation for avionics, automotive, railway, and industry applications. SCADE specifications are based on block-diagrams and hierarchical state-machine graphical models with rigorous formal specifications. The SCADE KCG compiler is certified at the highest level of avionics certification, which suppresses the need for generated code unit testing. The SCADE tool has support for visual animation, test-suite coverage analysis, and formal verification. It has gateways to many other tools ranging from system-level specification to performance analysis.

1 Introduction

We describe the SCADE synchronous methodology and toolset dedicated to model-based embedded software design, validation, and implementation for avionics, automotive, railway, and industry applications. The overall idea is to generate correct-by-construction embeddable implementation from high-level executable formal specifications, increasing software quality while decreasing design and validation costs. Since the specification is executable, it can be thoroughly simulated and verified before embedding. Since the implementation is automatically generated, there are no errors introduced at the implementation phase.

The synchronous methodology is rooted in 25 years of scientific research [3,5,10,11] and 20 years of successful industrial application. It is based upon a conceptual model of embedded computation backed by four strong technical cores: specific high-level rigorous graphical and textual languages, formal semantics, compiling algorithms for correct-by-construction implementation, and formal testing and verification techniques.

SCADE evolved from the SAGA tool that was originally developed in 1986 by Schneider Electric [4] for nuclear plant safety systems, as a graphical version of the Lustre synchronous language of Caspi and Halbwachs [10]. It was then developed further to gradually replace the Airbus SAO internal tool for airborne software. It is now used by a large number of avionics, railway, industry, and automotive companies for fly-by-wire, engine control, brake control, safety control, power control, alarm handling, etc. SCADE embodies the KCG compiler from high-level designs to C that is itself certifiable at avionics DO-178B norm highest level A. TÜV certification is also available for automotive.

Source code development is based upon the Scade[1] graphical block-diagram notation familiar to control engineers, complemented by hierarchical Safe State Machines to describe state- or mode-oriented computations. These specification-level notations have precise mathematical semantics. Besides making software development more rigorous, they ease communication between engineers and between suppliers and customers. Functional verification is performed in two ways: conventional simulation techniques enhanced by graphical animation of the design and model coverage analysis, and formal verification of safety properties by model checking or abstract interpretation. Functional verification is needed only at block-diagram level, since the embeddable C code generated by the certified KCG compiler is automatically correct and qualifiable.

The rest of the paper is organized as follows. Section 2 discusses concurrency and determinism issues for embedded systems and introduces the cycle-based computation model. Section 3 presents the Scade block-diagram and state-machine formalisms; Section 4 discusses the formal synchronous semantics. Section 5 presents the software design and validation flow associated with SCADE, as well as the associated tool ecosystem. We give a brief comparison with conventional OS-based techniques in Section 6. We conclude in Section 7.

2 Concurrency and determinism of embedded software

Embedded software applications are very different from classical IT or networking applications. Instead of dealing with data files and asynchronous interrupts, they deal with the control of physical phenomena through specific sensor-actuator feedback loops and man-machine interfaces. Programs are mostly implementations of control algorithms. This calls for specific description paradigms close to the ones used in control engineering and systems engineering to design such algorithm: block diagrams for continuous control and state machines for discrete control. SCADE is directly based on these design paradigms. A comparison with principle with conventional techniques will be given in Section 6.

2.1 The need for concurrency

Concurrency is essential for all embedded applications. Control algorithms are most often built by assembling basic elements: samplers, integrators, filters, comparators, state machines, etc. These concurrent elements communicate by exchanging information on a time- or event-trigger basis. Here, concurrency means cooperation. This constrats with competition-based concurrency found in operating systems or thread-based applications where concurrent processes or threads compete to access and utilize resources.

[1] We use SCADE for the development environment and Scade for the base specification formalisms.

2.2 The need for determinism and predictability

Functional determinism is a key requirement of most embedded applications. A system is deterministic if it always reacts in the same way to the same inputs occurring with the same timing. On the contrary, a non-deterministic system is allowed to react in different ways to the same inputs, actual reaction depending on internal choices or computation timings. It is obvious that determinism is a must to control a car or a plane: the car should not decide by itself to go right or left. The same applies to man/machine interface or alarm handling.

Of course, deteterminism may not be a relevant requirement for other application types. An Internet connection naturally behaves in a non-deterministic way, and there is nothing wrong about that. In the same way, a car entertainment system may have some local non-deterministic behavior. But one does not control a car with an Internet-like infrastructure. This is why Scade specifications are deterministic by construction, the SCADE tools preserving determinism all along the specification-to-implementation chain.

On the implementation side, performance predictability is key to ensure functional determinism. In particular, one must ensure that internal computation timings cannot interfere with the functional timings of the control algorithm proper. Predictability is never an easy subject, in particular because of uncertainty due to caching and speculation optimization in recent microprocessors. But, of course, any form of added unessential non-determinism makes it even harder. SCADE achieves predictability by generating simple sequential code from concurrent specification using techniques described below.

2.3 The cycle-based concurrent computation model

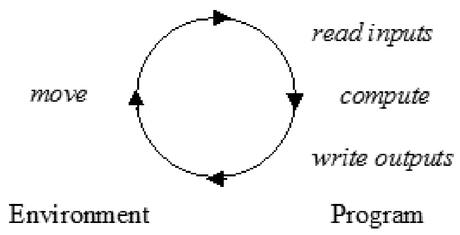

Fig. 1. Cycle-based computation

Cycle-based computation used by SCADE is a Folk model introduced long ago in many industrial designs to deal with embedded computation, but largely ignored by mainstream computer science. It consists of performing a continuous loop of the form pictured in Figure 1. In each cycle of this loop, there is a strict alternation between environment actions and program actions. Once the input sensors are read, the programs starts computing the cycle outputs and its own memory state change. During that time, the program is blind to environment changes, ensuring interference-freedom. When the outputs are ready, or at a given time determined by a clock, the output values are fed back to the environment, and the program waits for the start of the next cycle.

Things are very much as in a two-game play: players play in strict alternation and each player does not interfere with the other player's thinking. The cycle-based model can also be viewed as a direct computer implementation of the ubiquitous sampling-actuating model of control engineering and signal processing.

In the implementation, there are several ways to control the cycle: time-triggered computation starts the cycle on a regular basis; polling consists of restarting the cycle as soon as it is over, and event-triggered computation consists of starting the cycle whenever some even occurs. This is application-dependent and will not be detailed further here.

2.4 Synchronous communication and its realization by cycle fusion

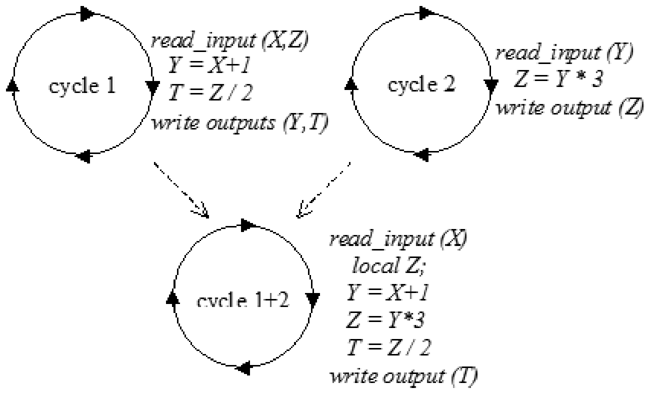

Fig. 2. Cycle fusion

In the cycle-based model, concurrent components communicate by exchanging information during the cycle. An output computed by a component is instantly broadcasted to all concurrent components that want to read it. If one wants to implement delayed communication, one can insert elementary delay components that output at each cycle their input at previous cycle.

Consider the concurrent cyclic components 1 an 2 in Figure 2, which specifies a non-trivial dialogue pattern. The first component reads X and Z and writes

Y and T, while the second component reads Y and writes Z. Communication is conceptually instantaneous: within a single cycle, Y is computed by 1 using X and communicated from 1 to 2, which causes Z to be computed by 2 and communicated back to 1. Communication is performed by a logically synchronous chain reaction, all enclosed within a single cycle representing a logical instant.

In a typical implementation, each component generates a straight-line code to execute the actions of its cycle. Communication between concurrent components it is realized in a very simple way by cycle fusion, see Figure 2: one merges the statements generated by the concurrent blocks blocks into a single straight-line code for the global cycle. Communication between the individual tasks is performed implicitly, by an adequate interleaving of the statements that respects inter-cycle communication dependencies. Notice that there is no overhead for communication, which is implemented using well-controlled shared variables without any context switching. For large hierarchical designs, cycle fusion goes across concurrent blocks and across the hierarchy, building a single sequential code from a network of components. Large-scale cycle fusion is unfeasible by hand but it is a relatively easy task for an automatic code generator. The SCADE compiler fully automates it and guarantees correct access to the shared memory.

Note that cycle fusion can be extended to support full separate compiling of blocks under some output-delay conditions not detailed here.

2.5 Determinism and predictability of cycle-based applications

Determinism is respected by construction, whatever the number of concurrent processes may be. Performance predictability is made relatively simple by cycle fusion, since the generated code is purely sequential and does not imply context switches. It is limited only by the intrinsic non-determinism of modern microprocessors due to cache access and speculative execution.

Notice that the cycle-based computation model carefully distinguishes between logical concurrency and physical concurrency. The application is described in terms of locally cyclic and logically concurrent activities. Such logical concurrency makes the designer's work much easier by breaking complex tasks into simple ones that communicate in a simple way. However, the implementation uses a single process at run-time. The Scade model can be extended to support multi-process execution and physical distribution of multiple processors, see [9], but the SCADE tool does not support this yet.

3 The Scade formalisms

For cycle-based design, SCADE provides the user with two familiar specification formalisms: block diagrams for continuous control and hierarchical safe state machines (SSMs) for discrete control. Both formalisms share the same view of a computation cycle and communicate in the same way.

3.1 Block diagrams for continuous control

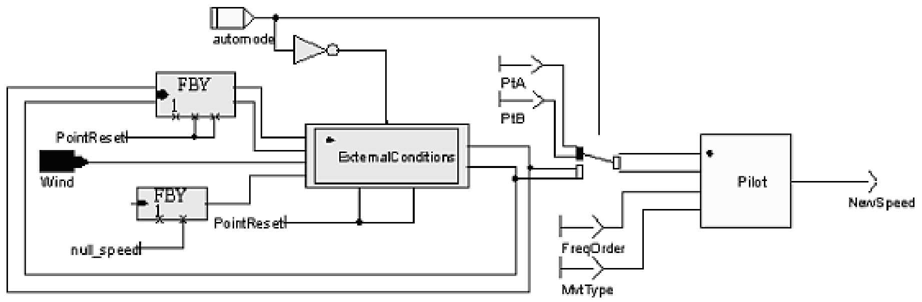

Fig. 3. Scade block diagram

By continuous control, we mean sampling sensors at regular time intervals, performing signal processing computations on their values, and outputting values, computed for instance using possibly complex mathematical formulae. Sampled data is continuously subject to the same transformation. In Scade, continuous control is graphically specified using block diagrams such as the one depicted in Figure 3.

Boxes are called nodes. They are concurrent objects that compute outputs as functions of inputs, with possibly internal memory. Arrow between nodes denote communication channels also called flows. They can carry data of any type. All nodes share the same cycle and only communicate through the arrows. In a cycle, communication is conceptually instantaneous: a data element sent by a node reaches its destination in the same cycle. Primitive delay nodes such as the *FBY* nodes in Figure 3 are available to beak synchrony. At initial cycle, an *FBY* node ouputs its initial value. Then, at each cycle, it outputs the value of its input at previous cycles. Any loop in the block diagram must contain at leat one delay element.

To add some flexibility in functioning modes control, Boolean flows can be used to control the activation of nodes. When a node N is controlled by an activation condition Boolean flow b, N is activated in a cycle only if b is true in the cycle.

Scade blocks are fully hierarchical: blocks at a description level can themselves be composed of smaller blocks interconnected by local flows. In Figure 3, the *ExternalConditions* block is hierarchical, and one can zoom into it with the editor. The same base cycle is shared by all the hierarchical components. A Boolean activation condition for a hierarchical node recursively acts on all its sub-nodes. Scade block hierarchy is purely architectural. At compile-time a hierarchical block occurring in a higher-level block is simply replaced by its contents, conceptually removing its boundaries, and cycle fusion is peformed on the whole flattened result. Therefore, there is no need for complex and often partial hierarchical evaluation rules often found in other hierarchical block diagrams formalisms.

Hierarchy makes it possible to break design complexity using a divide-and-conquer approach and to easily reuse library blocks. There is no need to write complex blocks directly in C or ADA, since defining them hierarchically from smaller blocks is semantically better defined, much more readable, and just as efficient.

3.2 Safe State Machines for discrete control

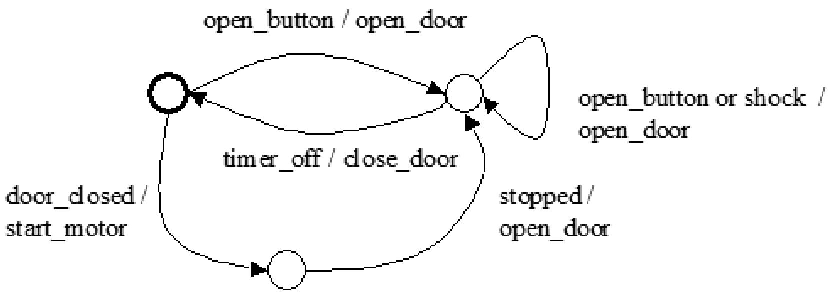

Fig. 4. Standard flat state machine diagram

By discrete control, we mean changing behavior according to external events originating either from discrete environment input events or from internal program events, e.g., value threshold detection. Discrete control is where the behavior keeps changing, a characteristics of modal human-machine interface, display control, alarm handling, complex functioning mode handling, or communication protocols.

Manually adding control Boolean flows and operations to block diagrams becomes rapidly messy when discrete control is non-trivial. One must resort to another well-known formalism: state machines. A standard flat state machine is pictured in Figure 4. As for a block diagram, it is composed of boxes, arrows, and names, but with a different meaning: boxes mean states, arrows mean transitions between states, and names denote signals exchanged with the environment. In a transition label I/O, I denotes a trigger signal and O denotes a result signal. If the start state of the transition is active and I occurs, the transition is fired and O is emitted.

Flat state machines have been very extensively studied in the last 50 years, and their theory is well-understood. However, in practice, they are not adequate

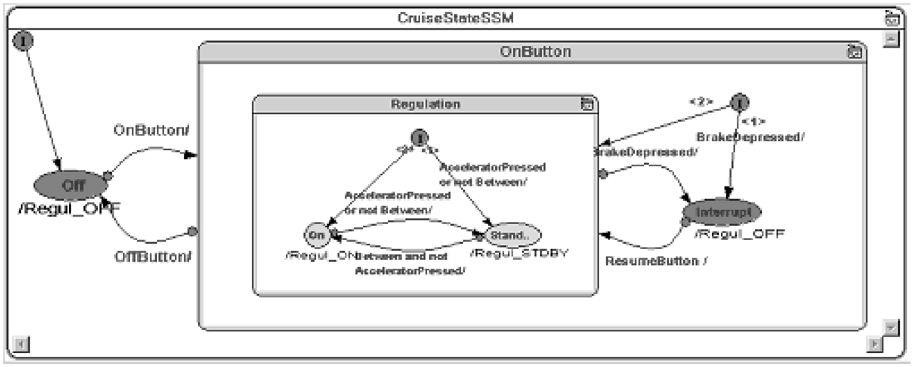

Fig. 5. A SSM hierachical state machine

even for medium-size applications, since their size and complexity tends to explode very rapidly. For this reason, richer concept of hierarchical state machines have been introduced, the initial one being Statecharts [12]. The Scade state machines are called Safe State Machines (SSMs), see Figure 5 for an example. These evolved from the Esterel programming language [5] and the SyncCharts synchronous statecharts model [2]. SSMs have been proved to be scalable to large control systems.

SSMs are hierarchical and concurrent. States can be either simple states or macrostates, themselves recursively containing a full SSM or a concurrent product of SSMs. When a macrostate is active, so is the SSMs it contains. When a macrostate is exited by taking a transition out of its boundary, the macrostate is exited and all the active SSMs it contains are preempted whichever state they were in. Concurrent state machines communicate by exchanging signals, which may be scoped to the macrostate that contains them.

The definition of SSMs carefully forbids dubious constructs found in other hierarchical state machine formalisms: transitions crossing macrostate boundaries, transitions that can be taken half-way and then backtracked, etc. These are non-modular, semantically hard to define, very hard to figure out, and therefore inappropriate for safety-critical designs. Their use is usually not recommended by most methodological guidelines anyway.

3.3 Mixed continuous / discrete control

Large applications contain cooperating continuous and discrete control parts. Scade makes it possible to seamlessly couple both data flow and state machine styles. One can include SSMs into block-diagram designs to compute and propagate functioning modes. Then, the discrete signals to which a SSM reacts and which it sends back are simply transformed back-and-forth into Boolean data flows in the block diagram on a per-cyle basis. The computation models are fully compatible.

3.4 Scade 6: full integration of block diagrams and SSMs

The above desciption is that of Scade version 5. The new Scade 6 formalism currently under development [8] will provide the user with a full interplay between block diagrams and state machines. In Scade 6, a state in a state machine may contain either another state machine or a block diagram. Two block diagrams enclosed in two exclusive states may refer to the same flow, making it possible to implement the mode automata described in [13], where one can switch from a continous control computation to another one for the same flows according to Boolean conditions.

4 Formal semantics

4.1 The formal synchronous semantics

The formal theory of synchronous concurrency has been developed in the last 25 years. It extends the cycle-based intuitive model into a fully precise synchronous computation model, which gives a strong theoretical basis to the compilation and verification of Scade programs.

We briefly illustrate the synchronous semantics using a very simple example in continuous control. We refer the reader to [5,10] for the formal development, more examples, and the handling of discrete control. Consider the following specification: given a discrete integer input flow I, output at each step the average A of the values received so far. In basic mathematics, one would use a discrete time index t and write the following system of iterative equations:

$$\begin{aligned} N_0 &= 1 \\ N_{t+1} &= N_t + 1 \\ T_0 &= I_0 \\ T_{t+1} &= (T_t + I_{t+1}) \\ A_t &= T_t / I_t \end{aligned}$$

Such an equation system is good enough for mathematical reasoning, but not for software engineering that requires much more precision. In mathematical notation, one never cares too much about what is allowed or disallowed for indices, because the reader is assumed to be a technically skilled human being. Making A_{t+1} depend on A_{t+2} instead of A_t is a syntactically legal mistake that any reader readily detects. But computers are definitely unskilled and they faithfully reproduce any mistake. Therefore, we need a precise programming formalism in which such mistakes can be detected and rejected at compile-time. Lustre, the root textual language of Scade, was created for this purpose. In Lustre, the program is written below[2]:

[2] The equivalent Scade graphical program will not be pictured here. The SCADE compiler would actually translate it into the above Lustre textual form.

```
node O (I : int) outputs (A : float);
var N : int, T : int;
let
   N = 1->(pre(N)+1);
   T = I -> pre(T) + I;
   A = T / N;
tel;
```

The identifiers I, N, T, and A denote data flows, which are infinite sequence of values. For instance, the single identifier A represents the whole cycle-based infinite sequence of inputs $A_0, A_1, \ldots, A_t, \ldots$, where t denotes the cycle index in the computation, i.e., the logical time. Operators such as addition add sequences componentwise, i.e., in a synchronous way: $A+B$ is $A_0+B_0, A_1+B_1, \ldots$ The `pre` delay operator delays a sequence by one cycle: $\texttt{pre}(A)$ is the sequence $-, A_0, A_1, \ldots, A_t, \ldots$, where the first element is left uninitialized. The '`->`' initialization operator returns its left operand at first cycle and its right operand at further cycles. Since it increments its previous value at each cycle, the N symbol denotes the sequence $1, 2, 3, \ldots$, T denotes the accumulated sum of the input values, and A denotes the required sequence of average values. The semantics of Lustre simply defines the sequences corresponding to the variables as the solutions of the system of equations. Here, when seen as a complete flow, N is indeed equal to $1- >(\texttt{pre}(N)+1)$. The Lustre and Scade formalisms and semantics extends to node activation conditions using a notion of derived clock, see [8,10].

Using the well-defined Lustre operators, the informal system of equation has been transformed into a fully rigorous program. By construction, there is no way to refer to N_{t+1} instead of N_{t-1}, since there is no operator returning a future value at any given instant.

4.2 Logical vs. physical time

In the synchronous approach, one counts logical time only in terms of I/O cycles. Synchrony simply states that events occurring in the program are viewed as logically simultaneous if and only if they occur in the same cycle. One only distinguishes between computations occurring in the same cycle and computations occurring in successive cycles. Therefore, at Scade specification level, the physical time it takes to perform an addition or a division is ignored. This is a basic separation of concerns principle: high-level specifications need not care early on about performance-related issues. However, computing on if physical time is required by the application, one can deal with it using an extra specific extra input.

5 The SCADE application development flow

5.1 The SCADE Y Development Cycle

The classical software development cycle is called the V cycle. Development flows down from systems requirements to embedded code, the lower tip of the V, while validation flows up from embedded code unitary tests to system-level functional tests. Three steps are particularly difficult and expensive in this cycle: the precise specification of software requirements in a specification language, their precise coding in an executable language, and the low-level testing phase, which is usually the most costly. For embedded software development, the SCADE process and tools help in four ways:

– At the specification level, the transition from mathematical simulation tools to fully precise programs suited for a qualifiable software flow is much more direct than with classical executable language hand-coding, thanks to he native block diagrams and state machine formalisms. This shortens the requirement-to-specification phase.
– Embeddable C code is automatically generated from Scade descriptions by the KCG compiler. For avionics, KCG is qualifiable at highest level A w.r.t. DO-178B guidelines. For automotive, KCG is certified by the TÜV Sud authority at SIL 3 level of the IEC 61508 standard and valid for the development of software up to SIL 4. Because of this, the object code can be considered correct-by-construction w.r.t. the source specification since the code generator itself is qualified with the very same process as the full application. The need for C-level unit testing vanishes.
– Since the Scade model is executable, functional verification can be performed earlier and better. This will be detailed below.
– Because of the intrinsic performance predictability of code generation by cycle-fusion, performance validation is also made easier. Abstract-interpretation based performance evaluation tools are very useful there, see [1].

Altogether, the V cycle is transformed into the Y where the junction between secification and inplementation is done at Scade specification level instead of C code level. The thin bottom of the Y represents certified code generation, now certified to be correct.

5.2 Model validation

Functional validation of an embedded system consist in checking that the system fulfills its requirements. Validation checks can be dynamic or static, as detailed below.

5.3 Dynamic checks and coverage analysis

Dynamic checks consist in test-suite based functional verification. A key issues is to build an appropriate test base, providing a large set of model inputs with minimal redundancy. This very notion is not easy to define. One usually use various coverage criteria to measure how much a test base stresses a model and how well it describes the possible input cases. SCADE uses an elaborate notion of model coverage, which includes exercising conditional nodes, reaching bounds on operators, etc. (See also the classical MCDC coverage requirements for Boolean expression covering [7]). Generating the test suites can itself be difficult. It can be done either manually or by extracting model boundary inputs from system-level simulations.

5.4 Static checks

Static compile-time checks consist in basic type-checking augmented by dead code detection and block diagram connections checking: absence of unconnected I/O pins and absence of cyclic data dependencies.

5.5 Formal verification

Assertion-based verification performs symbolic model-checking[3] to verify the validity of user-provided temporal assertions about program behavior. Assertions can either be derived from application requirements or correspond to self-consistency defensive programming checks developed during the software design phase. They are expressed in the Scade formalisms, technically as Boolean flows that should never become false. Thus, the user does not need to learn specific property-definition languages to use the verifier. Good examples of properties to show by model-checking are *regulation is active if in on state and if speed lies between 30 and 130 km/h* or *the elevator never travels with the door open*. Counter-examples for false properties are automatically generated. Notice that assertion-based verification is now routine in the hardware field. Its extension to cycle-based designs is natural since these are akin to "software circuits".

Abstract-interpretation based model checking checks for the absence of run-time arithmetic exceptions or array out-of-bounds access, see [6]. It is automatic and does not require the writing of assertions. It has been used successfully on very large avionics projects.

Formal verification techniques complement human testing abilities very well. In particular, they are very useful in finding nasty bugs that escape conventional testing but do show up in production systems. We believe that formal verification engines will continue making constant progress in the future and will become among the most efficient anti-bug weapons.

5.6 The SCADE automotive ecosystem

A tool never solves a problem by itself. Therefore, SCADE is coupled with many other tools acting in the systems design or software engineering areas. Designs

[3] SCADE uses the Prover plug-in verification engine from Prover Technologies.

can be imported from prototypes written in mathematical simulation environments using a semi-automatic importer. UML specifications can be linked to SCADE designs using gateways. Systems requirement are traced in the SCADE design using links with requirement management tools. Documentation is automatically generated from Scade designs.

The C code generated by SCADE for automotive applications is platform independent and MISRA compliant, which is essential for automotive applications. It only uses a small subset of C, with no dynamic memory allocation, no pointer arithmetic, and no loop, callable through a very simple API. for specific execution platforms, SCADE extends the API by providing a customizable wrapping technology that allows a straightforward integration in any target environment: wrapping to OSEK tasks or to popular RTOS tasks are available. Currently, within the AUTOSAR consortium, the generation of AUTOSAR compliant Basic Software Modules and Software Components with SCADE is studied. This includes the compliance with the merging automotive standard ISO 26262.

SCADE embodies other software engineering tools that intendx to make the development flow as smooth and safe as possible. The SCADE implementer tool makes it possible to finely control the fixed-point implementation of numerical computations for processors that do not support floating-point. Processor-dependent C compilers are checked to adequately compile the code generated by SCADE using a compiler verification kit that systematically compiles and checks all possible generated C patterns.

6 Comparison with operating-systems based designs

The other prominent model for embedded control is rooted in computer engineering tradition. It consists of writing sequential tasks for individual computations and using an operating system (OS) to schedule and run the tasks according to various criteria. The advantage is to rely on well-tested and robust off-the-shelf operating systems or language run-times. However, correctness issues become very application-dependent instead of being solved once for all by the programming formalism. The key issues are how the tasks are scheduled and how memory accesses are controlled.

Preemptive dynamic scheduling solves the problem at run-time in a generic way. However, it is a competitive concurrency model where tasks compete for resources (processor cycles, I/O, etc.). This interference-based model introduces a high level of nondeterminism and correctness is difficult to ensure. Shared memory accesses have to be controlled by semaphores or similar devices, know to be deadlock-eager and hard to check. Another potentially nasty problem is priority inversion, where a low-priority task permanently takes precedence over high-priority ones. For a specific system, one can show application-level determinism, predictability, and scheduling safety using fancy analysis techniques, but this is never simple.

To improve on this and provide a safer view of dynamic tasking, higher-level rendezvous concurrency primitives have been included conventional languages such as ADA. However, they still rely on OS-like mechanisms and consistency is equally challenging.

Another well-known technique is fixed static scheduling, using algorithms based on task durations, on deadlines, on priorities, etc. Tasks can be either preemptible by other tasks or non-preemptible. This works well for a relatively small number of components, with a reasonable preservation of determinism. However, compared to cycle-based design, static scheduling is more difficult to organize, hard to scale to large applications, and very sensitive to specification changes.

Altogether, the tasking model does not help the programmer in conceptualizing the problem. There is no consistent way to go from a V to a Y cycle.

Of course, cycle-based computation does not rule out the need for basic OS functions. An embedded OS is still needed to perform low-level functions such as communication with sensors and actuators through drivers. But this is far less complicated than full tasking, and the main cycle code remains deterministic and predictable. Notice that a little amount of non-determinism in reading sensor values or driving actuators at cycle boundaries may remain without danger: all robust control algorithms do tolerate slight variations in the actual sampling or actuating timing.

7 Conclusion

We have presented the SCADE methodology and toolset, which are based on the synchronous computation model for embedded control software. SCADE addresses the design flow from precise specification to embedded code generation. It is used in major industrial programs to generate qualifiable implementation code from high-level block diagrams and state machines familiar to control engineers. This implies a dramatic cost reduction in one of the most difficult and error-prone part of systems development cycle. The use of SCADE also makes the upper and lower part of the whole cycle easier: the input formalism is close to classical notations used in high-level modeling, while simplicity of the generated code makes verification and implementation performance analysis simpler. SCADE is widely used avionics, and is being used used for automotive applications such as braking systems, suspension systems, entertainment systems, alarm systems, etc.

The mathematical model of synchronous systems on which SCADE is based is instrumental. It guides the user in the specification process, strongly grounds program semantics, drives compiler development and certification, and makes formal verification of programs possible.

Acknowledgements: the author thanks the various people involved in the design, development, and usage of SCADE: P. Caspi and N. Halbwachs who created Lustre, J-L. Bergerand and E. Pilaud who created SAGA, C. André who created SyncCharts, F-X. Dormoy who developed SCADE, J-L. Colaço who is the main Scade semantics and compiler architect, and the whole SCADE team.

References

1. M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In *In SAS'96, Static Analysis Symposium, LNCS 1145*, pages 52–66. Springer, 1996.
2. C. André. Representation and analysis of reactive behaviors: A synchronous approach. In *Proc. CESA'96, IEEE-SMC, Lille, France*, 1996.
3. Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.
4. J.L. Bergerand and E. Pilaud. Saga: a software development environment for dependability in automatic control. In *Proc. Safecomp'88*. Pergamon Press, 1988.
5. Gérard Berry. The foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
6. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *In PLDI 2003 ACM SIGPLAN SIGSOFT Conference on Programming Language Design and Implementation, San Diego, California, USA*, pages 196–207, 2003.
7. J.J. Chilenski and S.P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, September 1994.
8. J-L. Colaço, B. Pagano, and M. Pouzet. A conservative extension of synchronous data-flow with state machines. In *Proc. Emsoft'05, New Jersey, USA*, 2005.
9. A. Girault. A survey of automatic distribution method for synchronous programs. In F. Maraninchi, M. Pouzet, and V. Roy, editors, *International Workshop on Synchronous Languages, Applications and Programs, SLAP'05*, ENTCS, Edinburgh, UK, April 2005. Elsevier Science.
10. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. In *Proceedings of the IEEE*, volume 79(9), pages 1305–1320, 1991.
11. Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer academic Publishers, 1993.
12. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8, 1987.
13. F. Maraninchi and Y. Rémond. Mode automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, pages 219–254, 2003.