SAS architecture: Verification Oriented Formal Modeling of Concrete Critical Systems*

Annie Ressouche
INRIA Orion Project
Sophia Antipolis, France
ar@sophia.inria.fr

Jean-Yves Tigli
I3S Laboratory CNRS UMR 6070
University of Nice Sophia Antipolis, France
tigli@essi.fr

Valérie Roy
Centre de Mathématiques Appliquées
Ecole des Mines de Paris, France
vr@cma.inria.fr

Daniel Cheung
ESSI
University of Nice Sophia Antipolis, France
cheung@essi.fr

Abstract — Concrete critical systems validation is a major challenge in any development process. Model checking verification offers exhaustive and automated validation. To apply this methodology, the specification of critical systems must be supported by a formal mathematical well-sounded model. To this end, we rely on synchronous language to model such systems and to ensure model existence we consider systems falling into a particular SAS architecture. SAS systems are composed by a synchronous main controller that manages autonomous asynchronous tasks. Abstraction methods provide us with the required synchronous model. We show how our approach suits wearable computer applications designing and how the modeling we propose meets the requirements of wearable computer validation.

Keywords: Model Checking, Finite-State Machine, Synchronous Asynchronous Approach, Wearable Computer

1 Introduction

The problem we address in this paper concerns the field of verification oriented formal modeling of concrete critical systems (whether this critical aspect arises from the functional, real time, resource availability or context of use sides of the problem). The specification of such systems must be supported by formal, mathematically-founded methods in order to be satisfactory and reliable. Model-checking verification offers exhaustive and automated verification methods [6]. It requires describing the systems behavior as finite-state machines.

Reactive languages are intended for a natural and reusable description of reactive systems (i.e., which are permanently reacting to their environment). These languages, be they declarative, data flow oriented (Lustre [7]), imperative, control dominated (Esterel [2]) or declarative, multi-clock (Signal [7]) are all compiled into finite-state machines and therefore offer a suitable framework for applying model-checking techniques. We show how the Esterel language is well suited for specifying the control parts of critical systems and provides us with the required characteristics: it is reactive and the synchrony hypothesis ensures us that it is deterministic (same reaction to similar entries). Its design offers appealing operators to express reactivity (wait and emit events, preemption, powerful exception mechanism, etc.). Moreover, it allows us to decompose a synchronous specification into several communicating entities, each of them dedicated to a specific purpose. Then we show how model-checking is suitable for critical system validation.

Any realistic application accepts different levels of description involving entities of various kinds which will be modeled differently: some via a controller (synchronous), some via a model specific to their sequential activity (asynchronous). But in such a setting, model-checking cannot help to verify whole general critical systems. To overpass this drawback, we propose the definition of synchronous/asynchronous (SAS) architecture enabling a global modeling of the system for verification purposes. Our architecture tends to set aside and regroup the interactions between asynchronous components. Indeed, these interactions are very often at the source of the systems complexity and difficulties in their modeling. It encompasses three levels: decisional, behavioral and functional. At the behavioral level we find asynchronous, autonomous, concurrent and non-interacting components. Each of these components presents a collection of activities which can be dynamically reconfigured. The modeling methods for the asynchronous components will hinge on how critical their role is within the whole system. Critical components will be modeled in our synchronous formalism by presenting their behavior as an
abstraction of reality, thus enabling formal verification tools to be applied. The real implementation of these components will have to satisfy the constraints resulting from the verification of their model. The decisional level consists in a collection of synchronous controllers managing the activities of the asynchronous components and their reconfiguration. Each synchronous controller is individually validated and the validation of the global system rests on a modularity characteristic of model-checking (known as assume-guarantee paradigm [6]). The functional level is made up of dedicated components (hardware or software), the interface of which only is significant.

The paper is organized as follows. The next section introduces the formal model we consider to verify SAS systems. In section 3 we define the SAS architecture. In Section 4, we apply SAS architecture specification to wearable computer applications. We show how wearable computer field fits the SAS architecture, and how this architecture meets the requirements of such applications through a dedicated dedicated computer example. Finally, in section 5, we discuss some related works and future one and conclude.

2 Formal Modeling and verification

In order to apply formal verification techniques on system, its specification must be supported by mathematically-founded methods that allows representing its behavior as a finite-state machine. The section 2.1 describes the requirements that this formal modeling impose to language design and how the reactive languages fulfill these requirements through some precise hypothesis. The section 2.2 shows how reactive languages are a suitable framework for applying model-checking techniques.

2.1 The Synchronous Model

Synchronous languages are devoted to a natural and reusable description of reactive systems (i.e., which are permanently reacting to environment stimuli). They appear to be robust and expressive, high-level mathematical formalisms to specify reactive parts of critical systems. In the aim to increase system reliability, it becomes now necessary to use, every where it applies, methods ensuring their safety. Verifying system properties implies naturally to be able to exhibit all the system possible behaviors. That is the framework offered by reactive synchronous languages: a reactive program will be compiled in the mathematical model of finite automata that allows the use of exhaustive verification techniques.

2.1.1 The Synchronous Hypothesis

As already mentioned, reactive systems are always ready to react to external stimuli. The reaction rhythm is environment driven and the system must work accurately at the environment speed. When giving to a system twice the same entry sequence, it must always react the same way and cause the same output sequences, this is needed to ensure determinism. As system becomes complex, its size increased and system description needs expressivity facilities like hierarchy and modularity. A system is naturally expressed as concurrent communicating subsystems, each one assuming a part of the global specifications. Verification techniques, like model-checking, apply to the formal model of finite automata. In the aim to apply these techniques on our system, we require compiling it in this formal sequential model. In order to conciliate these requirements (reactivity, determinism, modularity, finite-state machine compilation model), the synchronous approach do hypotheses.

Communications, between system and environment and between subsystems, are non-blocking communications done by instantaneous broadcast of events. This ensure that every subsystems have a coherent view of event status. Let us call the following sequence a reaction: entries are given to the system, it is activated, it reacts, it produces outputs and it becomes again ready to react. The sequence of system reactions describes a logical time and induces the notion of instant. We can then speak about the system logical clock. Input events cannot arrive and being taken into account as the system is currently proceeding to a reaction: this is the hypothesis of atomicity. Through this hypothesis, input and output events of a single reaction are considered as being synchronous, reaction duration is considered to be zero and event emission and reception are considered as being simultaneous.

2.1.2 The Synchronous Languages

The classical reactive synchronous languages are: the imperative language Estere[2] dedicated to program control-dominated systems; the declarative language Lustre[7] suitable to program data-flow oriented systems; the declarative language Signal[7] helpful to program systems requiring multi-clock description; the graphic language SyncCharts[1] based upon the Esterel semantics; the mode automata one[10] useful when systems behavior is compound of different modes related by transitions between them. The choice among these languages mainly depends on application types.

The imperative synchronous model (Estere like) seems very convenient to express the control dominated parts of critical systems. In this model, the parallelism is a purely logical operator that has been introduced in the language only for the sake of expressivity. It simplifies the action of programming by allowing the code to be structured in a very natural way. The code evolves more easily and validated subsystems can be reused in different setting. Signal is the single mean to achieve communication between subprograms or with the environment. The language operators express constructions like concurrency of statements, sequence of statements, signal emission and expectation, test of signal presence, mechanisms of exception, weak and strong abortion, non-recursive sub modules calls, etc. This synchronous model allows instantaneous reaction to signal presence and to signal absence. The instantaneous reaction to signal absence guarantees the atomicity paradigm. As a consequence, a program can react instantaneously to a signal absence by emitting it,
that leads clearly to inconsistency. Since the compilation relies on a well-sounded mathematical semantics, this induces the compilation of some programs to be impossible to establish due to causality cycles between signals. When compilation succeeds, the compiled code is sequential and could be represented as well as by finite automata than net lists. This compilation model is very interesting: it allows the use of formal verification of programs by model-checking.

2.2 Critical Systems Validation

Several approaches address system validation: testing, simulation, theorem provers and model checking techniques. The most widely used verification technique are extensive testing and simulation. They offer a first level of validation and are useful methods to detect errors. Nevertheless, they are not exhaustive and can miss significant errors in large designs even if formal methods offer automatic tests generation and improve their coverage. Although there has been considerable research on the use of theorem provers, this technique is time-consuming and often require a great deal of manual intervention. Existing tools in this field are proof-checkers and the proof is built by the user. Model checking is an alternative verification technique exhaustive and automatic. Typically, the user provides a model of its application and the specification to be checked. The model checking algorithm will either answer true to indicate that the model satisfies the specification or give a counterexample execution showing why the specification is false.

2.2.1 Model Checking

Model checking practice encompasses three main activities: first modeling the design into a given formalism, second express the property to be checked and finally interpret the result given by the model checking algorithm. The most popular model used by model checkers is simple automata or communicating networks of automata. Such a model is well suited to describe the set of all the executions of a system. To get the model of a design, synchronous languages can be used and their compilation automatically gives the model.

Temporal logic is a dedicated language to express the properties we want to check. With such a logic, we can formalize properties about the executions of a system; an execution being a sequence of reachable states of an automata. A property is expressed from input and output events of the system (for instance: start and stop signals, alarm, speed,...) or a boolean combination of these. Temporal operators allow to speak about the ordering of events in time without introducing time explicitly (for instance: "the next state satisfies property p", there is a future state where property q holds,...). In addition, quantifiers express properties for all the executions (universal quantifier) or for an execution at least (existential quantifier).

The major drawback of model checking approach is the sheer size of the models to be handled. However, the emergence of symbolic model-checking (implicit model representation) and the reliance on SAT methods (procedures for deciding whether propositional logic formulae are satisfied) guarantee scaling up model-checking scope. Hence, in new model checker as NuSMV, an efficient SAT-solver algorithm complement the symbolic approach.

2.2.2 Model Checking Issues

In model checking activity, it could be difficult to build the target model. On one hand, we have to face the state explosion problem. On the other hand, the design can involve some components not straightly expressible in an automata network setting as real type variables. Nevertheless, powerful techniques have been investigate to get automata model checker can deal with. Useful ones are the abstraction both on data and control and the observer automata method.

The abstraction method results in smaller automata preserving temporal logic formulae. Several techniques are: the abstraction over the control which consists in partitioning the state space to get a smaller automata. At the opposite, the abstraction over data which consists to simplify the range of the variables (from infinite types to finite ones). We use these abstraction methods to model our SAS architecture. The data and the asynchronous tasks of our designs have possibly infinite domains. Only an abstraction over data will allow to express property concerning variables. Particularly, infinite type variable domains will be partitioned into a finite number of relevant sub domains.

The observer automata method consists in listening non intrusively only the executions we are interesting in. The reachability of a state of the model consisting in the parallel operation between the observer and the design model give us some informations about an expected behavior. This powerful technique is quite natural since it allows to express the property in the same language than the design we want to check.

3 Synchronous / Asynchronous Architecture

Any realistic application holds several levels of description, that involve miscellaneous entities of different nature and that require different models: some will be modeled by a controller (synchronous), some others by different sequential behaviors (asynchronous). We define an architecture (SAS) dedicated to systems where synchronous and asynchronous components need to coexist, so as to allow the modeling and verification of the global system.

Most of the component-oriented software environments allow a developer to program the final application in the classical four-steps process: 1) assembling hardware components; 2) designing applicative (know-how) software components; 3) configuring and assembling all the software components; 4) writing a minimal invocation software for the application.

This approach provides for basic components reusability, and turns out to be satisfying for developing simple applications made of chained functional components. This methodology leads to a two-level software architecture: an - often
implicit - applicative scheme for an assembly of functional components. This applicative scheme consists in a set of data communications between components and the control of each component for the overall application coherence (the real decisional part of the application).

When facing the need for developing more and more elaborated, complete and flexible applications, the complexity of software development and design becomes problematic, by making validation very difficult and preventing from any formal verification.

### 3.1 Behavioral and Functional Levels

A Behavioral Component is a domain specific component or assembly of functional components that is strongly connected to the physical inputs and outputs of the system. In fact, a behavioral component is a chain of execution we do not interact with other behavioral components. This solution prevents new functionalities to alter the execution of the system. Thus behavioral components are independent entities of the critical system.

Each behavioral component is likely to exhibit a set of activities, each corresponding to a thread or task, exclusive with each other within a given behavioral component, and propagating a control flow to its interconnected functional components. Let us note that, for each behavioral component, a "NAC" (no activity) activity allows to freeze the component. Behavioral components are autonomous in that they do not interact directly with each other. This allows selecting any activity for each behavioral component without having to worry about any possible dependency between those components, as opposed to more classical approaches (see figure 2). The interactions that would though be necessary, such as access conflicts to resources of the critical system, are then deferred and managed by the decisional level.

We will deal separately with the special case of the critical behavioral components. Actually, precisely because they are critical, some components will be modeled in our synchronous formalism by representing their behavior as an abstraction of the real life, thus enabling formal verification. Our synchronous model, given its operators assume-guarantee property (compositionality), will allow us to check some properties in the global system. The actual implementation of these components will then of course need to comply with the constraints resulting from their ideal model's verification, and will often be developed a posteriori. On the other hand, the non-critical behavioral components will most of the time encapsulate some existing applicative scheme and functional components. They will appear in the architecture as a piloting interface of their activities.

The functional level is achieved through hardware and software dedicated functional components, whose only the interface is meaningful. The set of functional components remains to be designed (implemented or chosen into libraries), assembled and tested by the application developer. Functional components are the application lowest-level components.
3.2 Decision Level

The decisional level is intended to deal with the global activity of the system. It is the central part of the complex critical systems we are designing. It is made of two parts: the abstract model of its environment and the decision process himself. This process takes advantage of the previously described works on the imperative-synchronous reactive model allowing for assume-guarantee properties and formal verification by model-checking.

The abstract model of the decisional level environment is made of two separate sets: input data and output data. We will see that these data do not need the same processing.

The abstract model of the data describes the environment context. We consider as input, any data that represents the state of the global system: both the environment and the driven system. We represent the set of all possible input data as the following Cartesian product: \( I = W_0 \times \ldots \times W_l \), where \( W_i \) are the definition sets of the different input data.

Nevertheless, it is interesting and necessary for formal modeling issue, to reduce the number of the possible states of such a structure. Hence, we avoid to take into account the large number of all the data possible values but consider only a more restricted number of values corresponding to a meaningful evaluation of these values.

In our architecture a set of preprocessing reduces the input data cardinality. Finally the data \( w_i \) can then be represented by a variable with \( n(i) \) possible states. The induced events correspond to data modifications. The definition set of the input-data abstract model (see figure 3), is the Cartesian product: \( I = [1..n(0)] \times \ldots \times [1..n(l)] \).

The abstract model of the output data represents the activity of the system i.e. the current activities of all the behavioral components. Each component has a set of activity \( \{a^1_i, \ldots, a^{m(i)}_i\} \), \( a^j_i \) being the \( j \)th activity of the \( i \)th component. The Cartesian product (see figure 3) \( O = [1..m(l)] \times \ldots \times [1..m(p)] \) is the finite structure of all the system output data. The reduction of the output-data number of values, contrary to the input data, is not a significant problem here, because the number of the activities of a component is in general very limited. A component is attributed at least two activities "component active" (AC) and "component inactive" (NAC) and can possess one be two additional activities.

![Figure 3: Data abstract modeling](image)

4 Using SAS Architecture for Critical Systems

We apply our approach to specific critical systems known as "wearable computer" systems. As already mentioned, a "wearable computer" system is a multi usage system working in any environment for mobile user. This term does not gather only applications related to "communicating clothes" or "cognitive clothes" but it concerns also critical embedded applications dedicated to mobile user help (dive computer, medical computer,..). We illustrate our purpose on a specific wearable computer: a dive computer application. We intend to show that the SAS architecture fits wearable-computer application design and we discuss the benefit brought by specifying applications in such a way.

4.1 Example: a Dive Computer

A dive computer is an embedded system dedicated to assist a diver during his activity. The dive computer has different functions. It computes and displays all the information needed during a dive: the diver physical's state (air consumption, heart pulsing, tissue gas rate,...) and the dive important values (battery voltage, depth, pressure,...). It helps the diver during the nominal phases of his dive: decompression ceiling management (ceiling duration and depth holding). It detects the problems, alerts the diver, suggests him a behavior to follow and keeps on assisting him until he gets out of water. For instance, in case of an air consumption problem, the computer will propose an emergency lift back without ceiling, control the lift back and, once the diver out of water, advise for a stay in a decompression box. The computer must warn the diver when his dives are too close one from each other or when the diver intends to take the plane too shortly after a dive.

This application is specified as a SAS design and implemented using WCOMP software environment: a Rapid Application Development Toolkit for Wearable Computer based on Java [4]; synchronous parts being embedded inside java components(see figure 4).

![Figure 4: WComp embedding of dive computer implementation](image)
The decisional component is a synchronous supervisor that listens to an abstract version of the dive-computer input data and manages the underlying behavioral components. The dive-computer input-data come from several sensors worn by the diver: the ascent rate, the pressure, the depth, the date, the tank pressure, the breathing rate, the heart rate, the air consumption, and the battery status. To obtain the input-data abstract model, pre-processing performs an abstraction of the input data: From a given physical model of the diver, we compute the required ceilings and their respective duration; from the diver’s air consumption and his tank pressure, we evaluate the dive time remaining. The input-data abstract model is the set of the possibly valued input signals, the synchronous supervisor listens to. Some behavioral components display significant information for the diver, while some others record the history logs of given input values.

The decisional component is composed by concurrent modules, each of them being dedicated to a specific task. In this example, we will describe more precisely three sub-modules: the decompression ceiling holding, the lift back management (both in normal and emergency modes) and the display of decompression information. The ceiling holding component is switched on when the diver has to perform a ceiling and switched off as soon as the ceiling duration is over. The lift back component has two different activities: a normal activity launched when a lift back is required with respect to decompression ceilings (event cc_lifl_back_needed emitted) and an emergency one which implies a quick lift back without respecting any ceilings (event cc_lifl_back_emergency emitted). The decompression component is launched when a lift back is started and has two activities: in case of normal lift back, it just computes the decompression until the lift back ends (i.e. when water_off occurs). Otherwise in case of emergency lift back, it takes care of managing the diver because the lift back has been done without any ceiling stop.

4.2 Implementation of the Dive Computer

The decisional module is described using the SyncChart synchronous-reactive graphical language.
part concerns the lift-back and the ceiling-holding components put in parallel while the right part concerns the decompression component. We gather together the lift-back and the ceiling-holding components since they are both preempted by an emergency-lift-back event. As a consequence of the language semantics, the left part behaves as follows: cc-lift-back and cc-ceiling-holding subprograms run and as soon as emergency-lift-back event is received, they are both stopped and cc-lift-back-emergency and cc-ceiling-holding-off are emitted. The implementation of the three behavioral components is given in figures 7, 6, 8.

4.3 Checking Dive Computer Behavior

With such a description we can simulate the decision level behavior with dedicated tools around the SyncChart language. We also can check for safety properties using a model checker for SyncChart based on BDD methods. Our dive computer is designed to fulfill the requirement that a behavioral component cannot start a new activity without having been deactivated from its previous one. We express this property by an observer SyncChart program (figure 9) that emits failure when the property is violated (i.e. cc_decompression_manange is received after cc_decompression_compute and cc_decompression_off in between the 2 receptions). Hence, the property is verified when the global system composed of the dive computer and the observer can never emit the failure signal and it is verified.

Similarly, we check that each component having several activities cannot be launched with two different activities in same time. In order to perform this verification, we define an observer that emits failure either when cc_decompression_compute and cc_decompression_manange are both emitted or when it is the case for cc_lift_back_emergency and cc_lift_back_needed.

With the model checking prover, we can state that the failure signal is never emitted.

Moreover, to face the size explosion of models, the synchronous language approach allows us to rely on modular reasoning techniques. The operators of the Esterel language exhibit a compositional property with respect to equivalence (noted $\equiv$) or preorder simulation (noted $\preceq$) relations over programs. Let $P$ and $Q$ be two synchronous programs, $P \equiv Q$ (resp $P \preceq Q$) if there is a bisimulation equivalence [12] (resp a simulation relation [11] ) between their respective model automata. Both of these relations preserves temporal logic properties: assume that $P \models \phi$ and $P \equiv Q$ (or $Q \preceq P$) then $Q \models \phi$. This compositional property allows us to substitute, inside a design, a program by an equivalent or a smaller one. For instance, concerning the parallel operator, we have : $P \ll P Q \models \phi$ then $P \ll P \ll \ll P \ll P Q \models \phi$ when $Q \equiv Q_1$ or $Q_1 \preceq \ll P$. As a consequence, we can apply the assume-guarantee paradigm for compositional reasoning. This technique verifies each sub module of a larger module separately and deduces property for the whole module. Suppose that we have two programs $P$ and $Q$, we can guarantee that $P$ verifies a property under some assumptions related to $Q$ and we prove separately $Q$ under assumptions related to $P$. By combining the set of guarantee and assumed properties about $P$ and $Q$, it is possible to verify the correctness of $P \ll P Q$ without building its whole state space. For instance, on the example, we can prove that the module cc_decomposition does not emit both cc_decompression_manange and cc_decompression_compute events in the same reaction and we deduce that the global program cannot. This deduction holds because the checked property handle events disjoint from the other side of the global program parallel.

5 Discussion and Conclusion

Similar work has been done in the area of asynchronous system modeling. A real time extension to UML [13] allows to describe both real time or not large systems. It is really a powerful extension but its major drawback is the lack of mathematical semantics which does not allow the use of formal methods to make any proofs. On another hand, an extensive work has been done to model distributed systems composed by synchronous entities communicating asynchronously [8]. These systems are totally specified in a synchronous way since the asynchronous scheduler can be

\[ A \sim A_1 \text{ means that the set of executions(or traces) of } A \text{ is included in the one of } A_1. \]
described in a synchronous design. But, such globally asynchronous and locally synchronous design does not fit our application field because some of our components are truly asynchronous.

Reliability of concrete critical systems becomes a real challenge and requires their specification with formal and mathematically-founded methods. Model-checking approach is an appealing issue to solve the problem of critical systems verification. Nevertheless, it is efficient when applied to synchronous reactive designs but realistic critical systems are mostly a mix of asynchronous and synchronous parts. To cope with this paradigm, we propose the SAS architecture. This architecture enables a global modeling of critical systems to allow the use of model checking techniques. Such an architecture is composed by a decisional synchronous controller that manages asynchronous behavioral components driving functional entities. Asynchronous behavioral components does not communicate each others and the communication is done via the controller. They are specified by a synchronous model when they are critical. As a consequence, we get a synchronous model for critical application. We apply our approach to the specific critical system of the “wearable computer” applications. Particularly, we describe a dive computer system and we verify safety properties using model checking tool.

The industrial tool we use to implement the example in section 4 is ESTERELSTUDIO [14] that provides model checking facilities around SyncChart programs. The lack of this approach is the disconnection between modular facilities relying on language semantics and a global compilation mechanism. To bridge the gap, we intend to define another synchronous language, powerful enough to describe SAS architecture designs but offering a true compositionality ensuring both automatic modular verification with model-checking technique and a separated compilation into effective code. Moreover, we could use powerful model checkers as NuSMV which offers new methodology as NuSMV which offers new methodology on symbolic methods than for SAT-solvers usage. The price we will pay is the connection with the target model checker, but the game is worth the candle.

References


