A Synchronous Language for Modeling and Verifying Real Time and Embedded Systems

Thomas Kropf, Jürgen Ruf, Klaus Schneider and Markus Wild

Institut für Rechnerentwurf und Fehlertoleranz (Prof. Dr. Schmid)
Universität Karlsruhe
Postfach 6980, 76128 Karlsruhe, Deutschland
email: {Thomas.Kropf,Juergen.Ruf,Klaus.Schneider}@informatik.uni-karlsruhe.de
http://goethe.ira.uka.de

Abstract. This paper presents the new synchronous language PURR, the system description language of the verification system C@S. The language is mainly based on Esterel, but extends it in several ways. These extensions are necessary to model systems at a high level of abstraction that makes state-of-the-art verification techniques applicable. Moreover, our new language directly supports transformational verification: most steps necessary to make a system description suitable for a correctness proof with a certain decision procedure or a theorem prover can be done inside our language, avoiding conflicts occurring from different semantics of different languages.

1 Introduction

The formal verification of systems requires languages to formally model the system implementation and to describe specifications unambiguously. Correctness proofs can then be performed using a calculus or a decision procedure which work on both descriptions. The main emphasis of this paper lies on a new approach to describe and specify systems in a way which is better suited for a formal treatment than existent description languages.

State-of-the-art methods for circuit design are based on hardware description languages such as VHDL [10] or Verilog [16]. Both languages are dedicated to the design and the fast simulation of circuits at various levels of abstraction. However, these languages are less suited for formal verification. First, they both lack of a clear formal semantics. The semantics for these languages have been given in an informal manner that is not always clear and hence lead to discussions about the exact meaning of some constructs. Recently, different formal semantics have been proposed to solve this problem [3]. However, the event-driven simulation oriented semantics of these languages still makes verification difficult and they are less suited to describe systems at higher levels of abstraction.

A completely different approach is based on synchronous languages, which have been developed for the design of reactive embedded systems. In these languages, most statements do not consume time and those statements that do so are explicitly controlled by the user. Moreover, parallel threads run synchronously, i.e. both have the same ‘clocking scheme’. This view of threads simplifies the design of software for microcontrollers as well as hardware design for complex controllers to a large extent.

There are different synchronous languages such as Esterel [2, 7] that represents an imperative synchronous language, Lustre [15] a synchronous dataflow language, and graphical synchronous languages such as statecharts [5], Argos [8] or SyncCharts [4].

1This work has been partly financed by the DFG projects ‘Automated System Design’ and ‘Verification of Embedded Systems’ (see http://goethe.ira.uka.de/ for more information on the projects).
In contrast to hardware description languages like VHDL or Verilog, these languages have been designed with the application of formal methods in mind. The result have been clear, readable languages with a clean and formal semantics. The statements are as orthogonal as possible to achieve a small and understandable set of grammar rules. Moreover, the synchronous semantic model is excellently suited for the application of verification techniques, in particular for temporal logic model checking.

Hence, synchronous languages are an excellent starting point for a new language dedicated to formal verification. However, each of these languages lacks certain features which are essential for modeling and verifying a wide range of hardware systems as well as real time embedded systems at different levels of abstraction. To keep the advantages of synchronous languages without the restrictions hindering their intended use, our approach is based on defining a new system description and verification language called PURR by smoothly enriching the synchronous imperative language Esterel by the lacking language constructs. We have chosen Esterel as the basis of our approach as this language offers excellent possibilities to describe control dominated systems and there are many tools available for this language.

In the following, we will give a brief overview about our additions and motivate the decisions made. We will then present some aspects of PURR as well as some small examples. In contrast to Esterel, PURR offers the following additional features:

- nondeterminism
- abstract data types with polymorphism and dependent types [14, 1]
- generic modules
- special processes for finite state machines and net lists
- critical regions
- formal property specification and analysis with assertions

One of the main differences between Esterel and PURR is the use of nondeterminism in PURR. While traditional synchronous languages enforce deterministic systems, we allow and want to have nondeterminism as this is essential for modeling systems at higher abstraction levels. Consider for example, that a thread of a set of threads wanting to access a shared resource is to be chosen for arbitration. As long as the arbitration is fair and correct, we do not need to know how it actually works and can model the choice nondeterministically. Hence, nondeterminism may arise by various sorts of abstractions. Also, nondeterminism may arise in timing issues: due to abstractions, the delay or computation time of a component may often be determined only to a certain degree of exactness. Such timing behavior is modeled in PURR by timing intervals where the actual transition time is chosen nondeterministically from a specified interval.

Another major difference between Esterel and PURR is the availability of (polymorphic) abstract data types. Esterel only covers the control of parallel running threads while data types other than integers, floats, booleans and strings are imported from a ‘host language’ like C. Esterel can be translated afterwards into this host language such that a complete host language program can be created. From the designer’s point of view this is reasonable, since all the heuristics for the code optimization of already existing modern compilers can be used and only the code generation for the control of parallel running threads is left to the Esterel compiler. However, from the verifier’s point of view, this is a serious drawback, since an entire formal verification that should also cover the data manipulations is difficult to perform in such a language ‘mix’. Hence, we decided to extend Esterel by inductive definitions of abstract data types that are suitable for formal verification [9].

Subsequent differences between Esterel and PURR such as generic modules and syntax for directly describing hardware structures at register-transfer and synchronous gate level can be
viewed as ‘syntactic sugar’: They do not extend the expressiveness of the language as these constructs could in principle also be expressed in Esterel itself. However, as a main topic of C@S is the verification of hardware designs, it is convenient to be able to directly describe hardware structures and not to transform them to Esterel constructs. The possibility to describe generic structures allows the use of inductive proof methods to perform correctness proofs (also for abstract data types).

The most verification oriented difference of PURR to any other description language is the possibility to directly add specifications to the modules similar to the language of the SMV model checker. Thus it is not only possible to model the implementation behavior, but also to explicitly state the intended behavior. Thus PURR offers all necessary input to a formal verification, namely implementation and specification descriptions. In contrast to the SMV language, it is possible to give specifications also for submodules. Moreover, PURR allows to specify the behavior of the system environment, i.e. requirements about the external world which have to be fulfilled for a correct system functioning. This directly supports ‘assumption-commitment’ style correctness proofs.

In the same way as specifications, analysis requests are possible. While specifications of a module are either true or false, analysis requests describe problems for which a numeric solution is to be computed, e.g. the length of the shortest path between two states or two sets of states. Up to now, only timing analysis is supported, but there are many other applications, such as structural analysis to think about.

Besides the additional possibilities to model a wide variety of systems at different levels of abstraction, PURR directly reflects the transformation verification methodology of C@S: PURR allows to describe the behavior of the very same submodule in different models, e.g. as an algorithmic description, a timed finite state machine, a standard finite state machine or a netlist of simpler modules. Parts of a system may even be described by a set of properties only given e.g. as a temporal logic specification without having an implementation at all. This allows to perform the partitioning and transformation of implementation descriptions in PURR itself. These steps are necessary for the verification of complex systems, especially, if the correctness proof involves the application of decision procedures like model checking. As an example, an algorithmic description using only finite data types may be transformed into a finite state machine representation before it is fed into a model checker. It is only in the very last transformation step where the PURR description is exported in the file and language format of the target prover. Hence it is easily possible to add additional proof systems as backend modules of C@S. It has to be noted also that PURR has been carefully chosen in such a way that (at least the synchronous subset) of VHDL and Verilog can be translated into it. By providing such front ends for C@S a verification of synchronous hardware designs given in standard hardware description languages is also possible.

2 The Synchronous Language PURR

Due to the space restrictions of this paper it is not possible to present PURR or the C@S system in detail. In the following, we present only the basic ideas concerning abstract data types, the different types of modules as well as two illustrating examples.

2As far as possible, we aim at giving PURR a translational Esterel semantics, i.e. to define additional constructs in terms of Esterel programs such that arbitrary PURR descriptions can be translated into Esterel to use – besides our own verification environment – the available Esterel tools for synthesis and simulation.

3An overview of C@S and the complete grammar of PURR can be found at http://goethe.ira.uka.de/hvg/cats/.
2.1 Abstract Data Types

Abstract data types in PURR are built up by constructors, i.e. the set of values the abstract data type can take is isomorphic to the set of variable free terms build from the constructors \( c_1, \ldots, c_n \). Additionally, operators \( f_i \) can be defined on the abstract data type by induction on the constructors. The application of the functions \( f_i \), as well as the functions on already built-in types such as arithmetic operators on integers and boolean operators on boolean values, does not consume time.

\[
\text{ABSTYPE } \text{list}(\alpha) \in \text{NAT]}
\]

\[\text{CONSTRUCTOR}
\]

\[\text{NIL} : \text{list}(\alpha)[0];
\]

\[\text{CONS} : \alpha \# \text{list}(\alpha)[n] \to \text{list}(\alpha)[n + 1];
\]

\[\text{OPERATOR}
\]

\[\text{HD} : \text{list}(\alpha)[n] \to \alpha;
\]

\[\text{TL} : \text{list}(\alpha)[n] \to \text{list}(\alpha)[(n'?n - 1 : 0)];
\]

\[\text{AP} : \text{list}(\alpha)[n] \# \text{list}(\alpha)[m] \to \text{list}(\alpha)[n + m];
\]

\[\text{EQUATIONS}
\]

\[\text{HD}(\text{CONS}(x,y)) := x;
\]

\[\text{TL}(\text{NIL}) := \text{NIL};
\]

\[\text{TL}(\text{CONS}(x,y)) := y;
\]

\[\text{AP}(\text{NIL}, z) := z;
\]

\[\text{AP}(\text{CONS}(x,y), z) := \text{CONS}(x, \text{AP}(y,z));
\]

\[\text{END ABSTYPE}
\]

As an example, consider the above definition of polymorphic lists of a given length \( n \). All terms of that type are lists, where the members of the lists have type \( \alpha \) and the length of the lists is \( n \). Hence, we support polymorphism, as the use of type variables such as \( \alpha \) is allowed in other types. Each type can be substituted for \( \alpha \) such that these types serve in some way as ‘template types’. Also, we allow dependent types, as a type may depend on other terms. For example, the above type \( \text{list}(\alpha)[n] \) depends on the term variable \( n \) that determines the length of the lists. The usefulness of dependent types for hardware descriptions has be elaborated in [1] and needs not be outlined here. Also, we note without giving further details, that the inductive definition of data types as given above is well suited for a formal treatment, e.g. for correctness proofs by induction (see e.g. [9]).

2.2 Modules in PURR

In general, there are different kinds of modules in PURR that are all described similarly:

\[
\text{Module_Sort name} \ [g_1 : \gamma_1, \ldots, g_j : \gamma_j] \ (i_1 : \alpha_1, \ldots, i_n : \alpha_n) \to (o_1 : \beta_1, \ldots, o_m : \beta_m)
\]

\[\text{REQUIRES}
\]

\[r_1 : \phi_1, \ldots, r_\ell : \phi_\ell
\]

\[\text{Module_Body}
\]

\[s_1 : \psi_1, \ldots, s_u : \psi_u
\]

\[\text{SPECIFICATION}
\]

\[\alpha_1 : \chi_1, \ldots, \alpha_w : \chi_w
\]

\[\text{ANALYZE}
\]

\[\text{End Module_Sort}
\]

\text{Module_Sort} is one of FSM, STRUCTURE or PROCESS. The variables \( g_1, \ldots, g_j \) are the generic parameters of the module, \( i_1, \ldots, i_n \) are the inputs of the module and \( o_1, \ldots, o_m \) are the outputs of the module. \( g_k \) has the type \( \gamma_k \), \( i_k \) has the type \( \alpha_k \) and \( o_k \) has the type \( \beta_k \). Available
types are integers, booleans, enumeration types, arrays on types, records on types, unions on types and also user-defined abstract data types as shown in the last section.

The difference between the generic parameters $g_k$ and the inputs $i_k$ is that parameters $g_k$ do not depend on time while inputs $i_k$ do. Generic parameters are used for defining generic modules, e.g. a module may depend on the bitwidth of its inputs that is then used as generic parameter for the module.

Requirements allow to describe assumptions about the behavior of the environment, but do not affect the behavior the module. These requirements are used for the verification of the module, as in general the given specifications do only hold when the requirements hold. Also, requirements can be used for code generation as already done in the Esterel compilers.

Requirements as well as specifications of the module can be given in different logics. Currently, temporal logics [6] as CTL, CTL*, LTL and CCTL [11] are considered as well as subsets of first- and higher-order logics to cover specifications on abstract data types.

The Module Body depends on the Module Sort. We do not describe the simple cases for FSM and STRUCTURE as these are exactly the same as in the SHDL language as given in [12] (SHDL is a forerunner of PURR). Instead, we describe here the PROCESS modules. Roughly speaking, processes represent Esterel modules. Given that $\tau$ and $\gamma$ are expressions, $\sigma$ a signal expression, $x$ a (signal) variable, $\alpha$ a type and $S_i$ a statement, the following statements belonging to Esterel are also available in PURR:

1. NOTHING, HALT and PAUSE $\tau$
2. $x := \tau$ and $x := \tau$ AFTER $\gamma$
3. VAR $x := \tau : \alpha$ IN $S$ END and SIGNAL $x := \tau : \alpha$ IN $S$ END
4. EMIT $x$ and EMIT $x(\tau)$
5. SUSTAIN $x$ and SUSTAIN $x(\tau)$
6. LOOP $S$ END
7. REPEAT $\tau$ TIMES $S$ END and POSITIVE REPEAT $\tau$ TIMES $S$ END
8. PRESENT $\sigma$ THEN $S_1$ ELSE $S_2$ END
9. IF $\tau$ THEN $S_1$ ELSE $S_2$ END
10. AWAIT $\sigma$
11. ABORT $S_1$ WHEN $\sigma$ END and ABORT $S_1$ WHEN $\sigma$ DO $S_2$ END
12. WEAK ABORT $S_1$ WHEN $\sigma$ END and WEAK ABORT $S_1$ WHEN $\sigma$ DO $S_2$ END
13. LOOP $S$ EACH $\sigma$
14. EVERY $\sigma$ DO $S$ END
15. SUSPEND $S$ WHEN $\sigma$
16. TRAP $x_1, \ldots, x_n$ IN $S$
   HANDLE $x_1$ DO $S_1$
   \ldots
   HANDLE $x_n$ DO $S_n$ END
17. EXIT $x$
18. RUN $m [\tau_1, \ldots, \tau_m] (\sigma_1, \ldots, \sigma_m)$
19. WHILE $\tau$ DO $S$ END
20. $S_1 ; S_2$
21. $S_1 \parallel S_2$
The meaning of the statements is almost what they say and is, exactly the same as in Esterel. For this reason, a discussion of them is left out here. See e.g. the Esterel Primer\(^4\) for an excellent introduction on Esterel. We have excluded all Esterel statements such as \texttt{call} and \texttt{exec} that have to do with the integration on tasks written in the host language, as \textbf{PURR} has no need for a host language.

In contrast to Esterel, \textbf{PURR} offers the possibility to extend the \texttt{pause} statement by an interval such that it consumes an amount of time units specified by that interval. The actual value of time units to be consumed is chosen nondeterministically of the interval. The \texttt{pause} statement in Esterel does always consume one unit of time. This allows timing nondeterminism.

The \texttt{RUN} statement allows to start the execution already defined module \(m\). The parameters \([\tau_1, \ldots, \tau_n]\) are thereby instantiations for the generic parameters, while the arguments \(\pi_1, \ldots, \pi_m\) are time-dependent inputs of the module \(m\).

In addition to Esterel statements, \textbf{PURR} offers two further statements: \texttt{REGION} \(x\ \mathcal{S}\ \text{END}\) allows parallel threads to share common data, which then is safely accessed in such a critical region via mutual exclusion. A region \(\mathcal{S}\) is shared between different threads through its name \(x\). Moreover, for any (temporal) logic formula \(\psi\), \texttt{ASSERT} \((\ell, \psi)\) is a statement in \textbf{PURR}. These assertion statements make it possible to specify properties which must hold at certain points of the computation flow. It introduces a label \(\ell\) to name that control point of the thread and adds the specification that the formula \(\varphi\) must hold as that point (hence the specification that always \(\ell \to \varphi\) must hold). The properties \(\varphi\) are given in the same way as in specifications, which are described before. This allows to reason about control points and their relationship as given in [17].

### 2.3 Examples

We will explain the remaining module types of \textbf{PURR} by two examples. A first example describes an \(n\)-bit-carry-ripple-adder (\texttt{CR}). This \textbf{PURR} program defines a stateless FSM module which implements a two-bit-full-adder (\texttt{FADD}). Based on this module, the carry-ripple-adder is described recursively by a generic construct. This generic description contains two cases, a terminal case using the full-adder as the sole component, and a recursive case using two components, a carry-ripple-adder with \(n - 1\) inputs and a full-adder. These two components build the carry-ripple-adder for \(n\) inputs.

Example 1 uses \texttt{FSM} and \texttt{STRUCTURE} modules. But only a few of the available features these module types provide are used in the example. A \texttt{FSM} module may additionally include states and a description for transition relations. The \texttt{STRUCTURE} module used in the example contains a \texttt{GENERIC} structure description, which allows to distinguish several different netlists for different values of parameters. It is possible to mix parameter independent netlists (as shown in next example) with \texttt{GENERIC} structure description in one \texttt{STRUCTURE} module.

Esterel allows only to describe deterministic behavior. We already pointed out, that the \texttt{PAUSE} statement introduces some kind of nondeterminism for the amount of time consumption. \textbf{PURR} offers also a more general kind of nondeterminism that is based on the choice operator [14]\(^4\); given that \(\Phi\) is any expression with a free variable \(x\) of type \(\alpha\), then \(\exists x : \alpha. \Phi\) is an expression that chooses an arbitrary value of type \(\alpha\) that satisfies \(\Phi\). Depending on that, value tests in conditionals or loops, etc. become nondeterministic to some extent. The choice operator is usually used in higher-order logics as e.g. HOL [14] and gives a very useful form of nondeterminism. Whenever a choice of some given set has to be made where the algorithm for making the choice is irrelevant, we can abstract away from the algorithm with a choice-expression.

\(^4\)\texttt{http://zenon.inria.fr/meije/esterel/}.\n
The second PURR program describes a producer-consumer communication via an external buffer module. The main data structure of the buffer module is given by an abstract datatype, which realizes a FILO-queue (figure 2). The producer process chooses nondeterministically a natural number and sends it to the buffer (figure 3). The nondeterministic choice is given by the the choice operator. This production consumes \( Z_{KD} \) time steps. The variable \( Z_{KD} \) is of type \( \text{TIME} \). This datatype describes time scopes or time intervals of natural numbers. Since this variable is instantiated to an interval \([8, 18] \) the production time is not fixed but out of the range 8 to 18.

The shared_buffer process receives natural numbers of the producer and pushes them into its local queue. If the consumer is able to receive a new value, it sends a signal \( \text{get} \) to the shared_buffer. If the buffer is not busy (writing received data of the consumer) and the queue is not empty, it sends the oldest queue value to the consumer and removes it from the queue by using the \( \text{pop} \) operation of the abstract datatype \( \text{BUFFER} \).

Also the consumer takes time to process the received data. This is expressed by the \( \text{PAUSE} \) statement. Since the \( \text{constime} \) variable is instantiated to a fixed value, there is no nondeterministic behavior in this module.
3 Conclusions

We have presented PURR a new language targeted at describing and verifying complex systems. It is based to some extent on the synchronous language Esterel, but enriches this by many constructs that are necessary and useful for modeling and verifying reactive systems as e.g. hardware circuits. We have already implemented the core part of C@S in C++ which is responsible for reading and manipulating PURR programs. Also, we have defined a formal semantics which is essentially for a language that is to be used for formal verification. This semantics is the basis for a deep language embedding in the HOL theorem prover that is now in order. Many verification algorithms and decision procedures such as CTL and LTL model checking are also currently integrated into C@S.
PROCESS\textit{producer}[\textit{proctime} : \textit{TIME}]() \rightarrow (\textit{result} : \textit{NAT})
\text{VAR}\ p : \textit{NAT};
\{
  \text{LOOP}
  p := \text{CHOOSE} \ p.\text{TRUE};
  \text{PAUSE} \textit{proctime};
  \text{EMIT} \textit{result}(p);
  \text{PAUSE} 1
  \text{END}
\}
\text{END PROCESS}

PROCESS\textit{consumer}[\textit{constime} : \textit{TIME}](\textit{data} : \textit{NAT}) \rightarrow (\textit{get})
\text{VAR}\ c : \textit{NAT};
\{
  \text{LOOP}
  \text{PAUSE} 1;
  \text{EMIT} \textit{get};
  \text{PRESENT} \textit{data} \text{ THEN}
  \text{PAUSE} \textit{constime}
  \text{END}
\}
\text{END PROCESS}

\text{STRUCTURE}\ \text{connection}() \rightarrow ()
\text{COMPONENT}
  \textit{prod} : \textit{producer}[[8, 18]];
  \textit{cons}(\textit{buf}.\textit{result}) : \textit{consumer}[10];
  \textit{buf}(\textit{prod}.\textit{result}, \textit{cons}.\textit{get}) : \textit{shared}\_\textit{buffer};
\text{END STRUCTURE}

Figure 3: Implementation of the producer and the consumer

\textbf{References}


