Abstract

Synchronous languages like Esterel and PURR have been introduced to support the development of concurrent reactive systems. These languages have an easily understood semantics that lends itself well for efficient implementations and also for a formal verification of the designs. We show how to enrich Esterel or PURR by two important features, namely asynchronous concurrency and nondeterminism. These features are important for (1) describing asynchronous systems and (2) for using PURR as a modeling language. We introduce the new constructs on the basis of a compilation oriented transition semantics of a core portion of PURR.

1 Introduction

1.1 Synchronous Programming

Many systems maintain an ongoing interaction with their environment and are forced to do this at the speed determined by the environment. Harel [15] therefore distinguishes transformational, interactive and reactive systems: Transformational systems read their inputs once at starting time, compute the outputs, and terminate. Interactive and reactive systems do normally not terminate. Instead, they have possibly infinitely many interactions with their environment where inputs are read and outputs are produced. A request from the environment is often called an action, and the computation of the desired output is called the reaction of the system. The difference between reactive and interactive systems is that the points of time where an interaction with a reactive system occurs, are determined by the environment, while the environment of an interactive system has to wait until the system is ready for a new computation. Hence, reactive systems must obey some real–time constraints. Therefore, they are typically implemented by more than one threads.

Synchronous languages [13] as Esterel [5, 6], PURR [17, 23] Reactive–C [9], SML [10], Lustre [14], SIGNAL [12], Statecharts [15], Argos [19], and SyncCharts [3] have been developed as special purpose languages for the design of reactive systems. The perfect synchrony paradigm of these languages considers the computation of reactions to be infinitely fast, so that the reaction appears at the same point of time when the action requests for it. This is achieved by an idealized view where the execution of most statements does not require time. Consumption of time must be explicitly programmed: Every atomic statement consume either none or exactly one unit of a logical time. Hence, by the semantics of these languages, all threads run synchronously to each other, since they automatically synchronize at the next time consuming statement.

Once a synchronous program has been written, its control flow can be compiled into a finite state machine. All manipulations of data values (that do not consume ‘logical’ time) are performed on the transitions of this finite state machine. Most compilers generate a sequential (C) program to implement that finite state machine as well as the corresponding data operations. As the semantics puts all statements that do not consume time on one transition, the number of states of the generated finite state machines is normally moderate. Also, the synchronous execution of threads yields deterministic systems, and avoids the state space explosion which normally occurs by the computation of product machines.

As the translation to a sequential finite state machine allows one to implement the multithreaded program by a single sequential process, one can even implement these systems without the process management of an operating system. Therefore, these languages are especially interesting for the programming of embedded systems, where often memory is rare. Moreover, there are techniques to directly map Esterel designs to register–transfer circuits [4]. It has been shown that the results of this hardware synthesis are almost optimal [25, 24], such that additional optimizations are usually not necessary. For this reason, Esterel can also be used as a good basis for hardware synthesis.

\[^1\]This work has been financed by the DFG project ‘Verification of embedded systems’ in the priority program ‘Design and design methodology of embedded systems’.
1.2 Asynchronous Programming

The perfect synchrony paradigm is, of course, only an idealization, since in any implementation either in form of a hardware circuit or a sequential (C) program, the execution of any statement will require some time. However, experience from hardware design has shown that synchronous systems can be designed much easier, and therefore most hardware circuits are nowadays synchronous. Berry compares this idealization with Newtonian mechanics: We know that Newtonian mechanics is only an idealization of the relativistic one due to Einstein, but the laws provided by the former are much easier to compute. Therefore, these rules allow one to predict much faster where objects are moving to. As the preciseness of these predictions is in most applications absolutely sufficient, relativistic effects can normally be neglected.

Analogously, the synchronous approach may be viewed as an idealization of the asynchronous one. However, the real world is sometimes absolutely asynchronous so that a synchronous idealization is questionable: In the design of synchronous hardware circuits, one must face the problem of clock skews which means that the clock signal (which is responsible for broadcasting the beginning of the next unit of the logical time) does not arrive at all locations on the chip at (physically) the same time. More generally, the implementation of distributed systems, where subsystems may be located at large distances, can certainly not be done in a synchronous manner.

Therefore, asynchronous extensions of synchronous languages are interesting for modeling and implementing distributed systems. For this reason, asynchronous extensions of synchronous languages have been given in [22, 1], and especially in [8], where communicating reactive processes are described. These processes are themselves implemented synchronously in Esterel, but run asynchronously to each other. The processes communicate via channels with each other instead of the usual broadcast communication of Esterel. However, it is not possible to integrate these asynchronous processes in a synchronous framework.

1.3 Combining Synchronous and Asynchronous Programming

Besides the application to model and to implement distributed systems, asynchronicity can also be viewed as a means to express that the execution order of certain threads is not important for the behavior of the system. With this view, asynchronicity gives the compiler the freedom to choose an arbitrary execution trace for the threads (cf. Figure 1) to optimize the generation of the sequential code, i.e., the finite state machine. In particular, one can achieve that data manipulating operations on the transitions are scheduled in this way so that, e.g., data operation units in hardware can be reused, lengths of clock cycles are reduced, and optimal load balancing of multiprocessor systems are achieved.

However, this requires that asynchronous execution must be integrated into the synchronous language, in that asynchronous processes may form a subsystem that runs synchronously to a surrounding synchronous framework. In this paper, we therefore describe an extension of Esterel that permits beneath the synchronous execution of threads also their asynchronous execution: While the synchronous parallel operator $||$ enforces each thread to proceed to the next time consuming statement, the asynchronous parallel operator $|||$ enforces only one thread to proceed, and permits the other ones to rest at their current state. In the asynchronous execution, it is however allowed that more than one thread proceeds at a time, which differs from an interleaved execution (where exactly one thread proceeds per time). Hence, asynchronous execution can be viewed as a generalization of both interleaved and synchronous executions, where each thread runs with its ‘own speed’.

Asynchronous threads are closely related with nondeterminism: While the synchronous execution of threads fixes for any input action exactly one reaction and exactly one next program state, the asynchronous execution of threads leads to several possible reactions and next program states. We therefore also consider the issue of nondeterminism, and introduce a special construct to explicitly program nondeterministic behavior.

The contribution of this paper is therefore to show how asynchronous concurrency and a form of nondeterministic execution can be added to synchronous languages like Esterel. We use the language PURR [17, 23], but our results can be generalized to any synchronous language. Moreover, we use a new form of transition semantics that is directly based on program locations. This semantics is compilation oriented, i.e., it can be directly used to implement a compiler.

The paper is organized as follows: In the next section, some basics on PURR’s syntax and semantics is presented. In Section 3, we present the new constructs for asynchronicity and nondeterminism. In Section 4, a compiler with these features and the issues of constructiveness and signal causality are discussed. Moreover, a brief overview of ongoing and future work is given.

2 Syntax and Semantics of PURR

In this section, we provide sufficient background information on PURR’s [17, 23] syntax and semantics. This semantics is interesting in its own right because it directly implements the finite state machine for the control flow. For this reason, it makes sense to use this
2.1 Syntax and Informal Semantics

In general, one distinguishes pure signals and signals that carry a value. Pure signals have a status, i.e., they are either present or absent. The current value of a signal is stored until it is changed; the status of a signal is however not stored: when time proceeds, the status becomes inactive. Such an emission is performed by the PURR statement `emit a`. When it starts execution, it broadcasts `a` throughout the scope of `a` and immediately terminates, since it does not consume time. Other processes within the scope of `a` could test its presence, e.g., by the by the statement `present a then S1 else S2 end`, where statement `S1` is executed if `a` is present, and otherwise `S2` is executed.

The following grammar gives an overview of the entire range of basic PURR statements this work is based on: (Curly brackets designate optional parts.)

```
proc ::= nothing | pause | emit signal | present expr then proc \{else proc\} end | proc; proc | proc || proc | \{weak\} while expr do proc end | \{weak\} suspend proc when \{immediate\} expr | \{weak\} abort proc when \{immediate\} expr | signal signal in proc end | [ proc ]
```

```
expr ::= true | false | signal | expr and expr | expr or expr | not expr | ( expr )
```

We only consider pure PURR, i.e., the fragment of the language that does not deal with data values. Note that this is only for reasons of conciseness in this paper, and not a restriction of the presented work. Hence, our expressions are only the boolean ones as given above.

We have already discussed the `present` and the `emit` statements. `nothing` is the empty statement, i.e., it does neither emit signals nor does it consume time. `pause` is the only basic statement that consumes time: When `pause` starts, it emits nothing, but its execution consumes one unit of time. Statements can be composed into other statements in many forms: `S1; S2` denotes the sequential composition, i.e., when it starts, it immediately executes `S1`, and if `S1` terminates at some point of time `t1`, `S2` is immediately started at `t1`. Note that if `S1` never terminates, `S2` will never be executed, and if neither `S1` nor `S2` consume time, then also `S1; S2` will instantaneously terminate.

If `B` is currently absent, `while B do S end` simply does nothing. Otherwise, it starts `S`, and after termination of `S`, the behavior repeats. Hence, `while B do S end` is the same as `present B then S end; while B do S end`. It is required that `S` consumes in each case at least one unit of time. `S` is not aborted when during the execution of `S` the condition `B` does not hold. Process abortion is implemented by `abort S when B`, which starts `S` and executes it as long as `B` is absent. The weak version of `abort` allows all emissions of `S` at abortion time to take place, while the strong version ignores these emissions.

The compound process `suspend S when B` implements process suspension: whenever during the execution of `S` the condition `B` holds, `S` is suspended until `not B` holds. If this is the case, the control flow proceeds with the execution of `S` from the place where it was when the suspension took place. `signal a in S end` declares a local signal `a` that is seen in the statement `S`. Delimiters `[ ... ]` are just used to deal with operator precedence: usually `||` has a higher priority than ‘;’, so that `S1; S2 || S3` is the same as `[S1; S2 || S3]`.

Synchronous concurrency is brought about by the `||`-operator: if `S1 || S2` starts, immediately `S1` and `S2` are started. If both `S1` and `S2` are executed, their execution proceeds in a synchronous manner since all statements of `S1` and `S2` are executed until the next `pause` statement is reached. For example, consider the statement:

```
[ emit a; pause; emit b ] || [ emit c; pause; emit d ]
```

will emit `a` and `c` at starting time, will then rest for one unit of time, and will then emit `b` and `d` and terminates at that point of time. Note that as long as either `S1` or `S2` is active, `S1 || S2` is active, i.e., `S1 || S2` terminates if both `S1` and `S2` terminates.

Due to the synchronous concurrency and the broadcasting of signal emissions, the threads may heavily interact with each other. For example, consider the statement:

```
emit a
```

```
(A) present a then emit b end
```

```
present a then emit c end
```

Here, `a` is emitted in the first thread and tested for its presence in the other threads at the same time. Hence, `b` and `c` are also emitted. Hence, the compound process emits `a`, `b` and `c` and terminates at the current point of time. Note that the same behavior is obtained when we replace `||` with ‘;’. The sequential composition `emit a; emit b` emits both `a` and `b` at the same moment, for instance.
All constructors that can be used to build compound processes do not require any time for bookkeeping. Also, boolean conditions are evaluated in no time. As a consequence, the statement in (A) instantaneously terminates.

### 2.2 The Transition Semantics

Various semantics for Esterel can be found in the literature. Most of them are transitional, that is, they map Esterel processes to sets of specific judgments [7]. In principle, every such judgment represents that the control flow may move from some start state to a some end state while producing some signal output, given that some condition holds.

It is therefore reasonable to work out a transition semantics for PURR. Consider again the process

\[
[\text{emit } a; \text{pause}; \text{emit } b ] \\
\parallel \\
[\text{emit } c; \text{pause}; \text{emit } d ]
\]

Its behavior is such that \( a \) and \( c \) are simultaneously emitted, whereupon the execution pauses for one clock tick, whereupon \( b \) and \( d \) are simultaneously emitted. To formalize this, we assign labels to each individual pause–process as follows:

\[
[\text{emit } a; l_1; \text{pause}; \text{emit } b ] \\
\parallel \\
[\text{emit } c; l_2; \text{pause}; \text{emit } d ]
\]

Then the control flow can be described as starting in an initial state and pausing at \( l_1 \) and \( l_2 \) before it reaches a final state. Moreover, we are interested in external behavior, that is, in signal emissions and in what signal conditions must hold for some transition to take place. In the example, all of that can be represented by the transition sequence

\[
\langle \text{in} \rangle \xrightarrow{a,c \text{ true}} \langle l_1, l_2 \rangle \xrightarrow{b,d \text{ true}} \langle \text{out} \rangle.
\]

The general transition format is

\[
\text{state}_1 \xrightarrow{\text{emitted signals}} \text{condition} \xrightarrow{} \text{state}_2.
\]

A judgment of this form means that it is possible to proceed from state\(_1\) to state\(_2\) while emitting the signals above the arrow, given that the condition beneath the arrow holds in the environment. States are of the form \( \langle \text{in} \rangle \), meaning that the process is about to start execution, or of the form \( \langle \text{out} \rangle \), meaning that the process has terminated, or of the form \( \langle l_1, \ldots, l_k \rangle \), meaning that the control flow has reached the pause locations labeled with \( l_1, \ldots, l_k \). Conditions are boolean formulas over signals. A condition holds in an environment of signals \( E \) if it evaluates to true when one interprets the signals \( E \) as true, and all others as false. It is never the case that a condition mentions any locally emitted signals. For example,

\[
\text{present } a \text{ then } \text{emit } b \text{ end } \parallel \text{emit } a
\]

has the only transition

\[
\langle \text{in} \rangle \xrightarrow{a,b \text{ true}} \langle \text{out} \rangle.
\]

Here, the semantics of the first sub–process consists of the transition

\[
\langle \text{in} \rangle \xrightarrow{b \text { true}} \langle \text{out} \rangle,
\]

since the emission of \( b \) depends on the presence of \( a \). The semantics of the second sub–process consists of the transition

\[
\langle \text{in} \rangle \xrightarrow{a \text { true}} \langle \text{out} \rangle.
\]

The semantics of the entire process is computed by combining these transitions according to the semantics of the \( \parallel \) operator. In consequence, the condition \( a \) becomes true due to the second statement. It is not any longer required from the environment.

It is indeed possible to take account of all constructs from Subsection 2.1 in this way. More details on that can be found in Section 4. In anticipation, we note that transition semantics for Esterel use process statements to represent what points the control flow has reached in any particular state. This approach goes back to Plotkin’s Structural Operational Semantics [21]. It is disadvantageous as far as semantics–based compilers are concerned, since handling process statements requires space and time. Handling pause–labels, as we do it, is obviously much more efficient. This situation is the reason why our transition semantics may be regarded as compilation oriented.

### 3 Adding Asynchronicity and Nondeterminism

In this section, we introduce our new statements for asynchronicity and nondeterminism in PURR. For didactic reasons, we start with the latter.

#### 3.1 Guarded Commands

PURR already contains nondeterminism in the form of the selection of a data value from a given range: Given a data type \( \alpha \), and a boolean condition \( \Phi \), the expression \( \varepsilon z : \alpha.\Phi \) chooses an arbitrary value of type \( \alpha \) that satisfies \( \Phi \). The additional statement for nondeterminism to be introduced now is directly situated on the control flow level. We have found that this level is often the
one where one is actually concerned with nondeterminism. This experience is supported by the fact that similar features have proved very useful in modeling asynchronous concurrent systems. A prime example is the asynchronous concurrent modeling language Promela, which underlies the SPIN model checker [16]. Specifically, a central feature of Promela is constituted by a looping nondeterministic guarded commands constructor. We want to enrich PURR by asynchronous concurrency, and we also want to use PURR as a modeling language. It is, therefore, very reasonable to add nondeterministic guarded commands too.

The syntax is as follows: (Curly brackets with a “+” designate one or more repetitions.)

$$
\text{proc} ::= \text{guard } \{ \text{case } \text{expr then proc} \}^+ \{ \text{else proc} \} \text{ end}
$$

The semantics is such that every sub-construct of the form \text{case } \text{expr then proc} forms a branch whose guard is constituted by the boolean expression. When a guarded command is entered, then all guards are evaluated. Among those branches whose guard evaluates to true exactly one is chosen and its process part is executed. If no guard is true, then the else–branch is executed, provided that there is one. If there is no explicit else–branch, then it is implicitly treated as else nothing.

Unlike guarded commands in Promela, this constructor is not looping: It terminates as soon as the executed branch terminates, provided that there is such a branch. If not, it terminates instantly right away. Also, just like all other constructors that can be used to build compound processes, it does not require any time for bookkeeping. In the final analysis, it is a nondeterministic form of present\_then\_else\_end. As an example, consider the following process:

```plaintext
signal a in
while not a do
  pause;
  guard
    case true then nothing
    case true then emit a
  end
end
end
```

This process consists of a while loop that is executed as long as the signal \(a\) is not emitted from within it. It is up to nondeterministic choice whether the signal is emitted or not. The result is a process that realizes an unpredictably long, possibly infinite, delay.

### 3.2 Asynchronicity

Asynchronicity often comes disguised in the form of the possibility of spawning concurrent processes. Esterel actually allows one to launch external tasks in this way. We want to integrate asynchronicity into the language itself but want to retain the constructor for synchronous concurrency as it is. We have, therefore, opted for introducing another explicit constructor \(||\), whose syntax is as follows:

$$
\text{proc} ::= \text{proc } || \text{proc}
$$

The novelty of asynchronous concurrent composition within the context of synchronous languages calls for a precise semantics. At this place, the transition semantics from the previous section comes into play. As an introductory example, reconsider the statement (A) where we replace the synchronous parallel operator with the asynchrony one.

$$
[\text{emit } a; \text{pause}; \text{emit } b ]

||

[\text{emit } c; \text{pause}; \text{emit } d ]
$$

Different semantics could be defined for \(||\). For a short discussion, consider Figure 1. The left hand side (I) gives the behavior of the statement if the threads are executed synchronously, i.e. the semantics of (A). (II) and (III) define different semantics of an asynchronous parallel operator. The difference between both is that (II) starts both threads \(S_1\) and \(S_2\) synchronously when \(S_1 || S_2\) is started, while (III) allows asynchronicity even at starting time. This means that when \(S_1 || S_2\) is started with the semantics of (III), it is sufficient to start one of the threads, and start the other one later on. Note that the semantics of (III) only requires that \(b\) is emitted sometimes after \(a\) and \(d\) is emitted sometimes after \(c\).

We have decided to give \(||\) the semantics according to (III). The problem is here that we must furthermore distinguish whether a thread has already been executed or has not yet been executed. For this reason, we introduce further locations \(t_k\) that mean that the sub-thread indexed by \(k\) has not yet been executed. Note that it is required that at least one of the threads proceeds with its execution.

As can be seen, the resulting finite state machine for the generation of sequential code yields many more states in comparison to the synchronous approach. Moreover, we obtain a nondeterministic behavior. As already pointed out, we view different execution traces that result from asynchronicity as ‘don’t cares’ so that the compiler can choose any of these traces for its code generation. This offers a lot of possibilities for an optimized code generation whenever the threads do not interact too closely with each other.

The general compilation scheme is as follows: Let \(P_1\) and \(P_2\) be statements. Suppose that the result of compiling \(P_i\) is a set of transitions over the locations \(L_i\) with initial location \(i_0\) and final location \(i_f\), \(i \in \{1, 2\}\). States of the transition system are subsets
of $L_i$. Without loss of generality, we may assume that $L_1$ and $L_2$ are disjoint sets. Then, the result of compiling $P_1 \parallel P_2$ is derived as a set of transitions over the locations $L_1 \cup L_2$. These transitions are generated in two steps according to the following scheme. The first step computes the following set of intermediate transitions, where $S_i \subseteq L_i$ holds (expr$[A \leftarrow B]$ denotes the replacement of each occurrence of $A$ in expr by $B$):

- $S_1 \cup S_2 \xrightarrow{A\text{ expr}} S'_1 \cup S_2$ whenever $S_1 \xrightarrow{A\text{ expr}} S'_1$ is a transition of $P_1$
- $S_1 \cup S_2 \xrightarrow{A\text{ expr}} S_1 \cup S'_2$ whenever $S_2 \xrightarrow{A\text{ expr}} S'_2$ is a transition of $P_2$
- $S_1 \cup S_2 \xrightarrow{A_1 \cup A_2\text{ expr}} S'_1 \cup S'_2$ whenever $S_i \xrightarrow{A_i\text{ expr}} S'_i$ is a transition of $P_i$, $i \in \{1, 2\}$, with expr := expr$_1[A_2 \leftarrow \text{true}]$ and expr$_2[A_1 \leftarrow \text{true}]$

The second step transforms these transitions as follows: First, all states of the form $\langle \text{in} \rangle$ and all states of the form $\langle \text{out1}, \text{out2} \rangle$ are replaced by the new initial state $\langle \text{in} \rangle$ and all states of the form $\langle \text{out1}, \text{out2} \rangle$ are replaced by the new final state $\langle \text{out} \rangle$. Moreover, all remaining state components in$_i$ are replaced by an appropriate thread label. To this end, two fresh indexes $k_1$ and $k_2$ are chosen. Then in$_i$ is replaced by $t_k$. Finally, all remaining state components of the form out$_i$ are discarded.

In sum, what we basically do consists of generating a kind of cartesian product of the compilations of $P_1$ and $P_2$. Very similar accounts of asynchronous parallel composition can be found in many other contexts (see e.g. [20]). We regard this fact as supporting evidence for our own approach.

If $S_1$ and $S_2$ are instantaneous statements, i.e., if their execution does not require time, then $S_1; S_2$ and $S_1 \parallel S_2$ has the same behavior. This would also hold for $S_2 \parallel S_2$ if we would have chosen the semantics according to II in Figure 1, but is not necessarily the case for the semantics we have chosen. In fact, $S_2 \parallel S_2$ may consume time, even if both $S_1$ and $S_2$ are instantaneous statements. The reason is that the control flow can now also rest in states that have one of the newly generated locations $t_k$. (The control flow may never rest at initial nor at final states).

For example, consider the statements emit $a$ and emit $b$. Their transitions are:

\[
\langle \text{in} \rangle \xrightarrow{a \text{ true}} \langle \text{out1} \rangle \text{ and } \langle \text{in} \rangle \xrightarrow{b \text{ true}} \langle \text{out2} \rangle
\]

The compound statement emit $a \parallel$ emit $b$ has three possible behaviors: (1) The first thread emits $a$, whereupon the second one emits $b$; (2) the first and the second thread emit $a$ and $b$, respectively, in the same instance; (3) the second process emits $b$, whereupon the first process emits $a$. This is obtained when we compute the following transitions according to the previously explained schema:

\[
\begin{align*}
\langle \text{in} \rangle &\xrightarrow{a \text{ true}} \langle \text{t2} \rangle \xrightarrow{b \text{ true}} \langle \text{out} \rangle \\
\langle \text{in} \rangle &\xrightarrow{a, b \text{ true}} \langle \text{out} \rangle \\
\langle \text{in} \rangle &\xrightarrow{b \text{ true}} \langle \text{t1} \rangle \xrightarrow{a \text{ true}} \langle \text{out} \rangle
\end{align*}
\]

While the second transition does not consume time, the first and the third will require one unit of time, since the control flow will rest at the locations $t_2$ and $t_1$, respectively.

### 4 Discussion and Future Work

In Section 2, we have introduced the format of a compilation oriented transition semantics for a core portion of PURR; in Section 3, we have shown that this format allows us also to compile asynchronous parallel composition when further locations are added (these become necessary since we allow asynchronicity also for starting time). We have also added nondeterministic guarded commands to PURR. The motivation has been to give the compiler freedom for optimizations whenever threads are not required to run synchronously, so
that more efficient code can be generated. Moreover, nondeterminism and asynchronicity both strengthen PURR’s capabilities as a modeling language.

We have implemented a prototype compiler for PURR with the presented statements in Prolog. As outlined here, it translates PURR processes to sets of transitions. Prolog allowed us to write the code directly along the semantics. Nevertheless, there are two particularly critical points that still need to be discussed: causality and constructiveness.

4.1 Causality

Causality is understood to be the causal dependency between signals. A prototypical problem with this issue is the presence of \( a \) in the example, only one transition remains possible once \( b \) is present and one where \( b \) is absent. In the Esterel world, however, this conclusion entails that the process has the undesirable property of being nondeterministic. In consequence, programs with cyclic signal dependencies are rejected by Esterel compilers. The matter is complicated by the fact that processes such as

\[
\text{(B) } \text{present } a \text{ then emit } a \text{ end}
\]

Here, the presence of \( a \) depends on itself. In other words, there is cyclic causality. One could conclude that the process has two logically consistent transitions, namely one where \( a \) is present and one where \( a \) is absent. In the Esterel world, however, this conclusion entails that the process has the undesirable property of being nondeterministic. In consequence, programs with cyclic signal dependencies are rejected by Esterel compilers. The matter is complicated by the fact that processes such as

\[
\text{(C) } \text{emit } a; \text{ present } a \text{ then emit } a \text{ end}
\]

are constructive. Here, this stipulation makes sense because the presence of \( a \) can be inferred from the unconditional \text{emit} statement.

This situation is potentially difficult for us because our compiler works strictly compositionally. We must, therefore, be able to compile (B) in a suitable manner, so as to be able to compile (C). In fact, however, our basic algorithms make everything work out. In the example, the result of compiling (B) yields the transitions \( \langle \text{in} \rangle \xrightarrow{a} \langle \text{out} \rangle \) and \( \langle \text{in} \rangle \xrightarrow{\text{not } a} \langle \text{out} \rangle \), while the result of compiling (C) gives a single transition:

\[
\langle \text{in} \rangle \xrightarrow{a} \langle \text{out} \rangle.
\]

The idea behind all of that is simple: We always generate the entire set of logically consistent transitions. In the example, only one transition remains possible once \text{present } a \text{ then emit } a \text{ end} is put behind \text{emit } a.

4.2 Constructiveness

We have already indicated that Esterel insist on deterministic programs. Another logical consistency condition that must never be violated is \text{reactivity}. Reactivity means that there is always an immediate response to every possible input; while determinism means that, depending on the state and the input, this response is always the same. In fact, most violations of these properties go back to cyclic signal dependencies. To preclude cyclic signal dependencies and some other forms of non–well–formedness, one introduces a certain well–formedness condition, which is called constructiveness. Esterel compilers generally reject non–constructive programs. The introduction of nondeterminism as a language feature entails that we cannot adopt this discipline unchanged in the first place.

It turns out that for our purposes, programs need not necessarily be constructive. For one, we must obviously permit at least some forms of nondeterminism. Moreover, it is sometimes the case that a program makes sense although it is not constructive. We do not want to reject such reasonable programs. Our overall solution consists of not rejecting any context–correct program. The compiler is, of course, downward compatible with Esterel, since it produces the intended result whenever the program is constructive. The crucial point is that many non–constructive but meaningful programs are also compiled in the intended way. For example, constructiveness demands that all \text{while}–loops be statically pausing. The following \text{while}–loop is not regarded as statically pausing or, in other words, it is not constructive (Labels appear in brackets. They do not belong to the code):

\[
\text{while } \text{true} \text{ do}
\]

\[
\text{present } a \text{ then}
\]

\[
\text{emit } b;\]

\[
\langle 1_1 \rangle \text{ pause}
\]

\[
\text{end;}
\]

\[
\text{present not } a \text{ then}
\]

\[
\text{emit } c;\]

\[
\langle 1_2 \rangle \text{ pause}
\]

\[
\text{end}
\]

It is obvious that this \text{while}–loop is dynamically constructive since either the control will rest at \( 1_1 \); or at \( 1_2 \); in each iteration. Our compiler detects such situations. In this example, it generates the following transitions:

\[
\langle \text{in} \rangle \xrightarrow{b} \langle 1_1 \rangle \quad \langle \text{in} \rangle \xrightarrow{c \text{ not } a} \langle 1_2 \rangle
\]

\[
\langle 1_1 \rangle \xrightarrow{b \text{ not } a} \langle 1_1 \rangle \quad \langle 1_1 \rangle \xrightarrow{c \text{ not } a} \langle 1_2 \rangle
\]

\[
\langle 1_2 \rangle \xrightarrow{b \text{ not } a} \langle 1_1 \rangle \quad \langle 1_2 \rangle \xrightarrow{c \text{ not } a} \langle 1_2 \rangle
\]

4.3 Future Work

We regard the compiler as the first step towards a model checker for PURR with asynchronicity and nondeterministic guarded commands. A fundamental problem in connection with model checking is that of state explosion. In the synchronous world, one commonly
tackles it by representing finite state machines symbolically. This approach is so powerful that it has been possible to handle state spaces with $10^{120}$ elements and more [11]. In the asynchronous world, so-called partial order reduction constitutes a main technique of coping with state explosion. Combinations of symbolic representations and partial order reduction in model checking purely asynchronous systems have been recently discussed [2, 18]. We expect that similar techniques will be useful in doing model checking within our hybrid framework.

References


