Synthesizing Safe State Machines from Esterel

Steffen Prochnow Claus Traulsen Reinhard von Hanxleden

Real-Time and Embedded Systems Group, Department of Computer Science
Christian-Albrechts-Universität Kiel, Olshausenstr. 40, D-24118 Kiel, Germany
{spr, ctr, rvh}@informatik.uni-kiel.de

Abstract

Esterel and Safe State Machines (SSMs) are synchronous languages dedicated to the modeling of embedded reactive systems. While Esterel is a textual language, SSMs are based on the graphical Statecharts formalism. Statecharts are often more intuitive to understand than their textual counterpart, and their animated simulation can help to visualize subtle behaviors of a program. However, in terms of editing speed, revision management, and meta-modeling, the textual nature of Esterel is advantageous. We present an approach to transform Esterel v5 programs into equivalent SSMs. This permits a design flow where the designer develops a system at the Esterel level, but uses a graphical browser and simulator to inspect and validate the system under development.

We synthesize SSMs in two phases. The first phase transforms an Esterel program into an equivalent SSM, using a structural translation that results in correct, but typically not very compact SSMs. The second phase iteratively applies optimization rules that aim to reduce the number of states, transitions and hierarchy levels to enhance readability of the SSM. As it turned out, this optimization is also useful for the traditional, manual design of SSMs. The complete transformation has been implemented in a prototypical modeling environment, which allows to demonstrate the practicality of this approach and the compactness of the generated SSMs.

Categories and Subject Descriptors C.3 [*SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS*]: Real-time and embedded systems; D.3.4 [*PROGRAMMING LANGUAGES*]: Processors—Compilers, Optimization; I.6.5 [*Model Development*]: Modeling methodologies

General Terms Design, Human Factors, Languages

Keywords Reactive systems, Esterel, Statecharts, Safe State Machines, SyncCharts, textual/graphical languages

1. Introduction

Reactive systems are systems that have permanent interaction with their environment. The execution of these systems is determined by their internal state and external stimuli. As a reaction, new stimuli and/or a new internal state are generated. To describe the behavior of reactive systems, the family of synchronous languages has been developed, including Esterel [7], Lustre [14] and Signal [13]. These languages offer numerous control flow primitives such as concurrency and preemption that are pertinent to reactive systems, and the *synchrony hypothesis* [7] gives a sound semantical basis to these languages. As an alternative to the aforementioned, textual languages, one may also develop reactive systems using graphical notations, such as Statecharts. The Statecharts formalism extends the classical formalism of finite-state machines and state transition diagrams by incorporating the notions of hierarchy, orthogonality, compound events, and a broadcast mechanism for communication between concurrent components. Statecharts provide an effective graphical notation, not only for the specification and design of reactive systems, but also for the simulation of the modeled system behavior. Since the original Statecharts proposal by Harel *et al.* [15], numerous dialects of Statecharts have been developed and Statecharts have also been incorporated into the Unified Modeling Language (UML) [18]. Today, Statecharts are supported by several commercial tools, *e. g.*, Matlab/Simulink/Stateflow [22], Statemate [15] or Rational Rose [20]. In this paper, we are particularly interested in the Safe State Machines (SSMs) [3] dialect of Statecharts, which is a graphical variant of Esterel.

Consequently the developer of reactive systems may choose between the textual and the graphical approach to specify systems. In principle, they offer the same expressiveness and the same level of abstraction. However, there are notable differences in terms of practical use, and both approaches have their advantages. To consider just four areas, we briefly compare them in terms of comprehensibility, editing speed, suitability for revision management, and the support of meta-modeling.

Comprehensibility A commonly touted advantage of graphical formalisms such as Statecharts is their intuitive use and the good level of overview they provide—according to the phrase "a picture is worth more than a thousand words." Especially for visualizing complex structures, Statecharts have advantages compared to textual programming languages. Here, the one-dimensional flow of text provides poor overview over the whole system, while Statecharts provide clustering elements such as states, hierarchy and orthogonality to structure Statechart components. However, the graphical modeling of realistic applications often results in very large and unmanageable graphics, severely compromising their readability and practical use. Textual programming languages can represent precise details very well, while visual languages are good for higher level context.

Editing Speed Entering textual programming code to specify a system is very efficient. Due to the linear text flow, the insertion and deletion of symbols and words is very simple. Regarding graphical models, the two-dimensional nature brings some editing complications. Before inserting elements, the modeler must often "make room" first. The deletion of elements is also tedious, because it often leaves distracting "holes." To handle this and to produce nice and readable graphics, adjacent elements have to be moved, while

LCTES'06 June 14–16, 2006, Ottawa, Ontario, Canada.

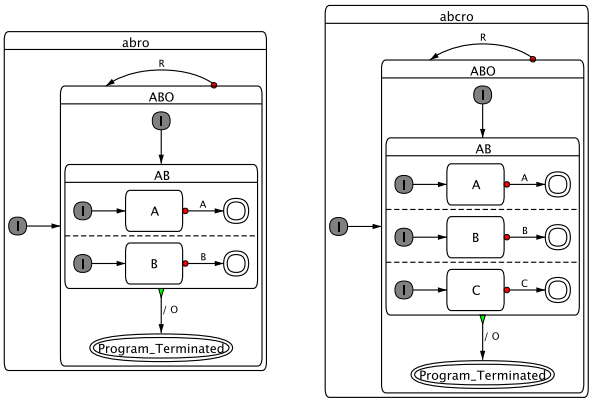
```
module ABRO:
input A, B, R;
output O;
loop
  [ await A || await B ];
  emit O;
each R
end module
```

(a) Original Esterel program

```
module ABCRO:
input A, B, R;
output O;
loop
  [ await A || await B || await C ];
  emit O;
each R
end module
```

(b) Extended Esterel program

Figure 1: Extending ABRO to ABCRO, Esterel versions



(a) Original SSM

(b) Extended SSM

Figure 2: Extending ABRO to ABCRO, SSM versions

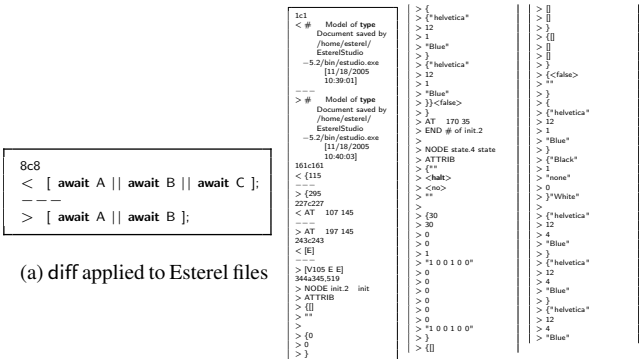
respecting the relationships of interacting elements (*e. g.*, transitions, state hierarchy, concurrency).

To illustrate this point, consider the canonical ABRO example [6], which is specified as follows: The system concurrently waits for two input signals, A and B. When both signals have occurred, the output O is emitted. This behavior is reset by the input signal R. An Esterel program expressing this behavior is shown in Figure 1a, an equivalent SSM is seen in Figure 2a. Now suppose we want to extend this example with a further accepting signal C in parallel to the signals A and B. To extend the Esterel version accordingly, into the program shown in Figure 1b, we just use our favorite text editor, move the cursor to the "]", and type "|| await C." Performing the same operation on the corresponding SSM, to obtain the SSM seen in Figure 2b, is rather more involved: we have to enlarge the top-level state and the states ABO and AB, we have to move the state Program_Terminated, we have to draw a new horizontal line, we have to draw the new state C and its predecessor and successor, we must draw two new transitions and label them—an operation likely to take an order of magnitude longer than the corresponding textual edit. Furthermore, the result is unlikely to be as precisely layouted as the one shown here, unless one applies further alignment operations—provided they exist in the modeling tool in the first place.

Revision Management When large repositories of textual code are developed, evolution is well traceable. At each milestone of a project, one can obtain revealing information about the increments of the programming work (*e. g.*, applying the UNIX diff utility to compare different versions). In general, there is no such possibility with graphical models. Even if the modeling tool stores models in a format to which tools such as diff are applicable (ASCII), these

```
8c8
<   [ await A || await B || await C ];
---
>   [ await A || await B ];
```

(a) diff applied to Esterel files



(b) diff applied to SSM files (scg format, as used by Esterel Studio)—only first 100 of 287 lines shown

Figure 3: Results of applying diff to compare ABRO with ABCRO, alternatively using the Esterel representation and the SSM representation

formats are not intended to be human-readable, and the relevant information is typically buried in irrelevant data.

To illustrate, Figure 3 shows the differences (produced by diff) between ABRO and ABCRO. For the Esterel representations, the difference file is 4 lines long, and we can immediately deduce how the programs differ. For the SSMs, the difference file is orders of magnitude larger and so cluttered that it is practically worthless to the human examiner.

Meta-Modeling One would often like to express models in a generic fashion, at an abstraction level that is higher than what is directly supported in the (textual or graphical) modeling language. One trivial example is to extend ABRO from two to n signals that are awaited. For another example, one might want to model a distributed system communicating via some protocol with n stations. (One such example is the Token Ring Arbiter used in the benchmarks in Section 5, with 3, 10, 50, and 100 stations.) It is generally rather easy to achieve this with textual languages that have standard macro capabilities, using generic scripting or preprocessing languages such as perl or m4, or just a powerful text editor. Graphical languages may also have macro capabilities, but creating a top-level system with n stations still requires manual work for each instance.

In summary, both textual and graphical languages have their specific domains and advantages. The traditional model-based design flow starts with the graphical entry of a system model, from which textual programs are synthesized; however, as we argue here, it would actually be advantageous to allow the designer to work in the opposite direction as well.

The main contributions of this paper are:

- A synthesis mechanism that derives a graphical Statechart model, specifically the SSMs dialect, from a textual imperative programming language, Esterel (Section 2);
- A set of optimization rules that transform a Statechart (SSM) into an equivalent, but more compact Statechart (Section 3)—these rules can also be useful to Statecharts developed the traditional way;
- A discussion of how to assert the correctness of such a transformation and optimization (Section 4);

- An experimental evaluation, based on an implementation of the aforementioned synthesis and optimization mechanisms in a modeling environment (Section 5).

Related Work

SyncCharts [1, 4], the predecessor of SSMs, have been defined via a translation to Esterel [2], and the commercial tool Esterel Studio generates Esterel programs from SSMs as part of its code generation process. To our knowledge, there have been no attempts yet to explore the other direction as we do here.

In general, there is only very limited work on synthesizing Statecharts from textual descriptions, and the approaches that we are aware of already assume that the textual input directly expresses the Statechart topology as an AND/OR tree (*e. g.* [16, 8]). This already achieves the advantages of textual entry, but does not offer the rich, concise control flow constructs available in synchronous programming languages such as Esterel.

As discussed in Section 2, one complication arising in the transformation presented here is to express Esterel's trap mechanism in SSMs. This situation also arises in the Esterel derivate Quartz [21], which does not support traps directly, see also Section 4. However, Quartz allows to test the current configuration of a system; as we do not have this option in SSMs, we present an alternative mechanism based on explicit trap signals.

Regarding Statechart optimizations, there have been already proposals for design rules that define "good" Statecharts; however, they typically focus on checking consistency [11] or graphical appearance, and do not attempt to systematically reduce the complexity of a Statechart as we do here.

2. Transforming Esterel

Esterel is an imperative synchronous language and consists of a set of kernel statements from which other statements are derived [5]. In contrast to other transformations and analyses on Esterel, we should not restrict ourself to these kernel statements, because the derived statements give us useful information about the structure of the program.

To aid the understanding of the transformation, let us briefly review some Esterel basics; for a more detailed reference, see Berry [6]. Different parts of an Esterel program communicate via *signals*, which have a boolean status. *Valued signals* can also carry an additional value, like an integer or real number, or a value of user defined type. The execution of Esterel programs is based on *instants*, and the execution of all statements is considered to take zero time and to take place within an instant, except for the pause statement, which explicitly waits for the next instant. As a consequence no signal can change its status during one instant. This is even true in the case of valued signals; multiple emissions of one valued signal in an instant can be combined by an associative and commutative function. The set of active input and output signals in one instant is called *event*. Important features of Esterel are the direct support of concurrency and multiple forms of preemption. Statements can either be *strongly aborted*, *weakly aborted*, *i. e.*, they are still executed during the abortion instant, or *suspended*, *i. e.*, they are frozen in this instant, but may resume later on. The preemption can either be *immediate* or *delayed*. In the immediate version, a statement can be preempted in the same instant it becomes active, while in the delayed version, it must be active for at least one instant before it can be preempted. Another important feature is the possibility to explicitly wait until a signal becomes present (or absent); this can also be either immediate or delayed. In addition to weak and strong abortion, Esterel provides *traps* as an exception mechanism. Traps are declared by the trap statement, which can optionally be augmented with one or more exception handlers. Traps are *raised* by the exit statement. We will discuss the behavior of traps in more detail in Section 4.

SSMs are a Statechart dialect with a synchronous semantics that strictly conforms to the Esterel semantics. A procedural definition of SSMs is given by André [3]. The basic object in SSMs is a *reactive cell*, which is a state with its outgoing transitions. Reactive cells are combined to *state-transition graphs*. A *macro-state* consists of one or more state-transition graphs. Additionally, SSMs can contain *textual macrostates*, which consist of plain Esterel code. States can also have *internal actions*: on entry, on exit and during. SSMs inherit the concept of signals and valued signals from Esterel. Hence a transition trigger can consist of an event, which tests for presence and absence of values, and a conditional, which may compare numerical values. Characteristic for SSMs are the different forms of preemption, expressed by different state transition types. Weak and strong abortion transitions as well as suspension can be applied to macrostates. A macrostate can either be left by an abortion, which has an explicit trigger, or by a *normal termination*, which is taken if the macrostate enters a terminal state. Analogously to Esterel, all transitions can either be immediate or delayed, where a delayed transitions is only taken if the source state was already active at the start of an instant. In contrast, immediate transitions may be taken as soon as the state becomes active; this makes it possible that one state is activated and deactivated multiple times within one instant. When a state has more than one outgoing transition, a unique *priority* is assigned to each of them, where lower numbers have higher priority. Weak abortions must have lower priority than strong abortions, and if a normal termination exists, it always has the lowest priority.

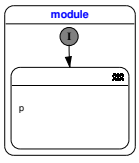
Our transformation from Esterel to SSMs is defined with a graph grammar, where non-terminal symbols are textual macrostates. These are transformed into graphical macrostates, which themselves may contain further textual macrostates. For each Esterel statement, one rule exists to transform it into one macrostate, where each sub-statement becomes a substate; thus the hierarchy of the SSM corresponds exactly to the nesting of statements in the Esterel program. A graphical macrostate enters a terminal state if and only if the corresponding Esterel statement terminates. Since Esterel and SSMs are closely related, the transformation of most language constructs is fairly straightforward. Care has to be taken to preserve the semantics of traps, which do not have a direct counterpart in SSMs.

We now illustrate the transformation scheme by first presenting the transformation rules that are needed for the ABRO example and then applying theses rules to the Esterel version of ABRO. In the following, p and q stand for arbitrary Esterel statements; de is an arbitrary delay expression, *i. e.*, an expression over signals plus the additional information whether this expression shall be evaluated immediately or in the next instant; s and t are arbitrary signal and trap names, respectively; exp is a signal value expression; ehe is an exception handler expression, *i. e.*, a boolean expression over trap signals. For details on the grammar of Esterel, see the Esterel manual [6].

Transformation Rule 1 (*module*)**.**

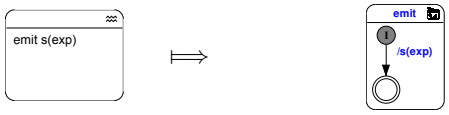
module mod_name:
  input $I1, ..., In$;
  output $O1, ..., Om$;
  p
end module

$\Longmapsto$



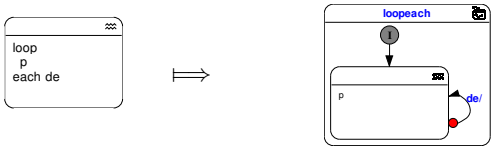
An Esterel program (a "module") starts with an interface declaration. Thus, the first step of the transformation is to generate an SSM with the same name and interface, which simply enters a textual macrostate that contains the program body p.

Transformation Rule 2 (*emit*)**.**



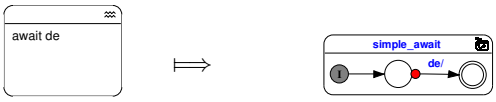
The emit statement broadcasts a signal s, with an optional value exp, and terminates instantaneously.

Transformation Rule 3 (*loop each*)**.**



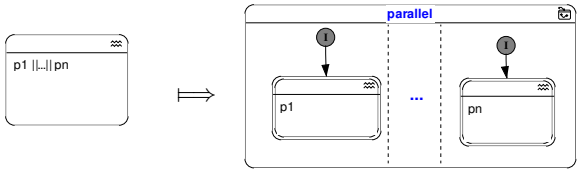
The loop p each de statement executes its body p and restarts it whenever the delay expression de is true.

Transformation Rule 4 (*simple await*)**.**



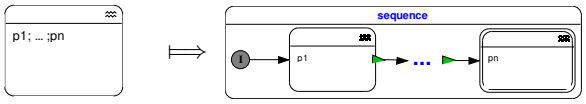
In its simple form, await de waits until the expression de evaluates to true and then terminates. The general case, not shown here, allows to wait for different events and execute specific code depending on which event occurred first.

Transformation Rule 5 (*parallel*)**.**



The parallel operators in Esterel and in SSMs work exactly the same. The parallel branches are executed synchronously. Note that the parallel terminates if all its sub-statements terminate; therefore, all contained macrostates are final.

Transformation Rule 6 (*sequence*)**.**



The sequence waits for termination of one statement before it starts executing the next statement. Note that a sequence terminates when its last sub-statement terminates.

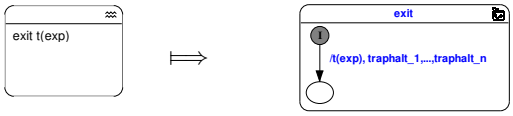
Transforming ABRO

The rules introduced so far suffice to generate an SSM for ABRO. The first step is to apply the (*module*) rule to the Esterel program that is to be transformed (Figure 1a), which results in an SSM with the same interface declaration as the Esterel program and a textual macrostate that contains the body of the given program. Figure 4 shows this SSM and the subsequent stepwise transformation into an SSM that contains no textual macro states anymore, hence no more transformation rules are applicable. The behavior of the original Esterel program is completely preserved in the resulting SSM; but we also see that the generated SSM contains unnecessary hierarchy nestings and is relatively hard to read. However, before considering optimizations in Section 3, we first consider some of the remaining transformation rules; due to space limitations, we do not present the complete set of rules.

Handling Traps

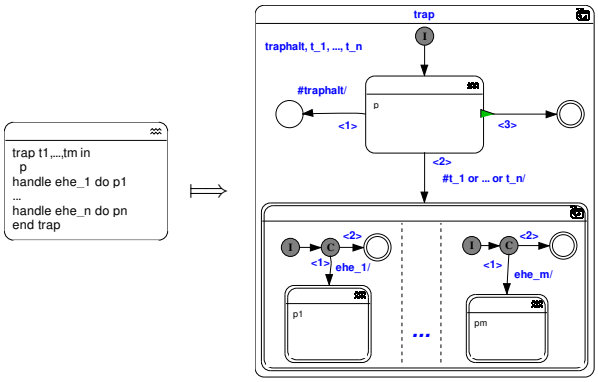
Traps are a part of Esterel whose translation to SSMs is not straightforward. Their behavior must be simulated using local signals and weak abortions. For each trap statement we introduce local signals for all declared exceptions, plus one new local signal traphalt. This signal is emitted if another trap with higher priority is activated from inside the trap scope. This is needed to assure that no exception handler is executed when an exception with higher priority is active at the same time; our translation handles the arbitrary nesting of traps. Whenever an exception is raised, all other exceptions inside its scope that are not running in parallel to the statement that raised the exception are deactivated. We will discuss this in detail in Section 4.

Transformation Rule 7 (*exit*)**.**



The exit statement raises an exception, optionally annotated with the value of an expression, which can be used in an exception handler. The SSM also includes signals to prevent the execution of exception handlers with lower priority (see also Section 4). Note that the exit state does not terminate normally, hence the simple state that is entered is not marked as final state. The body of a trap statement is weakly aborted when one of its exceptions is raised. When no other exception with higher priority is active, its active handlers are executed in parallel. When a trap with higher priority is active at the same time, no handler is executed. A trap terminates when its body terminates.

Transformation Rule 8 (*trap*)**.**



The trap catches exceptions which are raised inside its body. Depending on which exception was raised, different handlers are executed—possibly in parallel, when multiple exception were raised.

A requirement placed on Esterel programs is that they must not contain *instantaneous loops*; *i. e.*, the body of a loop is not allowed to terminate instantaneously. This requirement also transfers to SSMs; they must not contain instantaneous cycles. This causes a complication in our transformation of traps, since replacing traps by weak abortions can compromise the ability of a compiler (or simulator) to establish that there are no instantaneous loops in a given model. While for a trap the Esterel compiler analyses whether

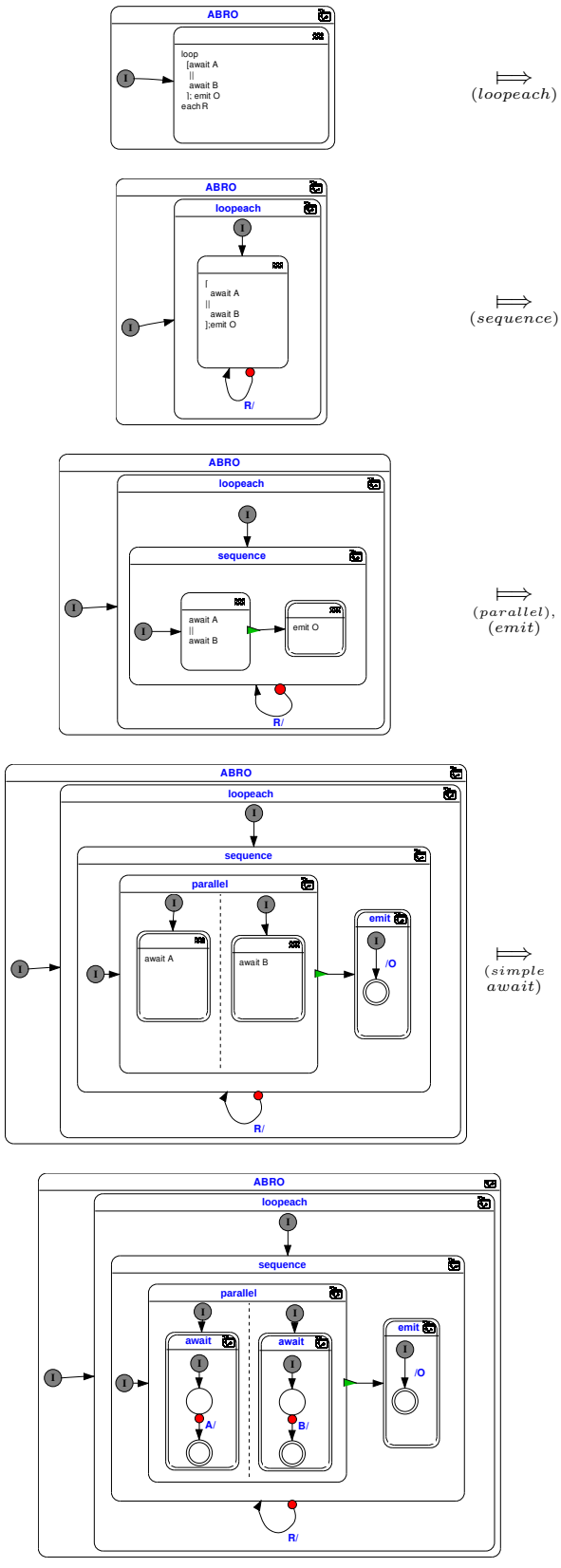


Figure 4: Stepwise transformation of the Esterel program ABRO (Figure 1a) into an equivalent SSM

it may be raised immediately or not, a weak immediate abortion is always assumed to be potentially instantaneous. A justification for this is that the scope of trap signals (*exceptions*) is confined to just the trap, whereas weak aborts may be triggered by signals with arbitrary scope. Consider the example shown in Figure 5a; here, the Esterel compiler determines correctly that within the loop, a pause statement must be executed before the exception T is raised, and that hence the loop is not instantaneous. However, if we would replace the trap with a weak abort, with an immediate trigger, the compiler would claim that the loop is potentially instantaneous and would reject the program—even though the program would be behaviorally equivalent.

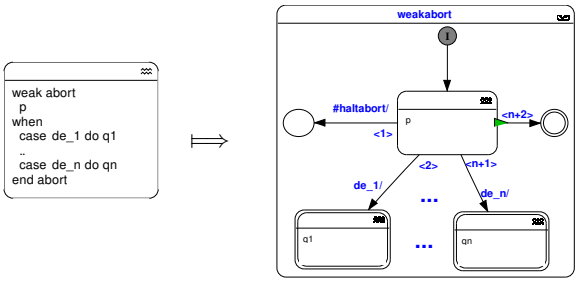
There are several possibilities to resolve this dilemma. One approach would be to apply an analysis to weak abort blocks with immediate triggers to determine whether the trigger signals can possibly be emitted in the instant when the abort block is entered; if not, we can safely replace the immediate trigger with a delayed trigger, which would solve the problem. This, however, would make the handling of the trap_halt signals and of traps with multiple handlers significantly more complicated. Another approach is based on the observation that for loops that are not instantaneous, we can safely add a parallel thread to each loop body that pauses for one instant (and does nothing else). This is illustrated in the Esterel module shown in Figure 5b, which is equivalent to the example in Figure 5a. The second thread within the loop, which just pauses for one instant and then terminates, does not change the behavior of the program, but allows the compiler to establish that the loop is not instantaneous, since parallel statements terminate only (normally) if all concurrent threads have terminated.

The first of these approaches has the advantage that it does not enlarge the program; however, we opted for the second approach, as it is conceptually simpler. Figure 5c shows the SSM synthesized from the Esterel example in Figure 5a, and we can see that the macrostate corresponding to the loop contains an extra thread that just pauses for one instant. However, to avoid excessive additions of such parallel threads, we limit this to loop bodies where it depends on an enclosed trap whether the loop is instantaneous or not, as is the case in this example. These cases seem to be rare in practice, in our benchmarks only ABCD contained such loops.

Handling Abortions

Abortions are closely related to traps, but since they can be directly expressed in SSMs, their transformation is much simpler. Both weak and strong abortion stop to execute their bodies when an abortion trigger is active. Depending on which trigger is active, different code parts can be executed.

Transformation Rule 9 (*weak abort*)**.**



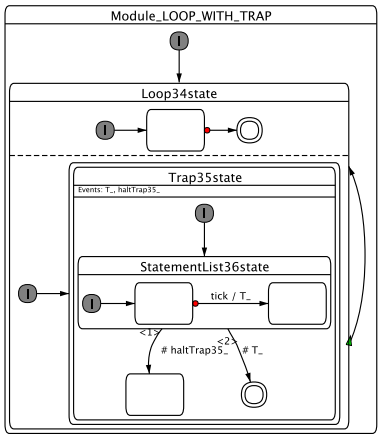
For weak abort, the body is still executed in the instant it is aborted. This makes it possible to raise an outer trap, which has priority over the abortion; the code that follows the weak abort is not executed. This is assured by the haltabort signal, which is raised by the exit statement similar to the traphalt signals.

```
module LOOP_WITH_TRAP:
loop
  trap T in
    pause;
    exit T
  end trap
end loop
end module
```

(a) Esterel program with trap

```
module LOOP_WEAK_ABORT:
loop
  signal T in
    weak abort
      pause;
      emit T
    when immediate T
  end signal
||
  pause
end loop
end module
```

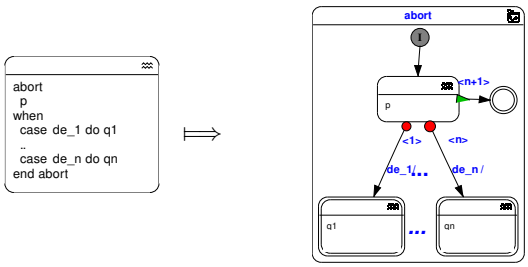
(b) Equivalent Esterel program with weak abort



(c) SSM synthesized for LOOP_WITH_TRAP

Figure 5: Handling of potentially instantaneous loops, introduced by the transformation from traps to weak abortions: the loop in the Esterel examples (a,b) is not instantaneous, which in the synthesized SSM (c) is made explicit with an extra parallel thread that is not instantaneous

Transformation Rule 10 (*abort*)**.**

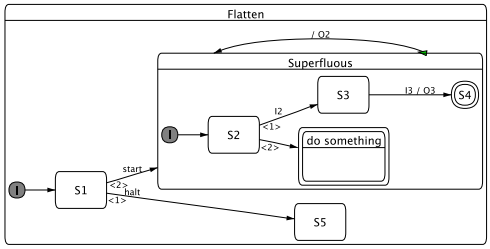


In contrast, for strong abortion no exception can be raised inside if the abortion condition is true, since the abortion body is not executed in this case. The same holds for other statements which are derived from strong abortion, such as loop each and every.

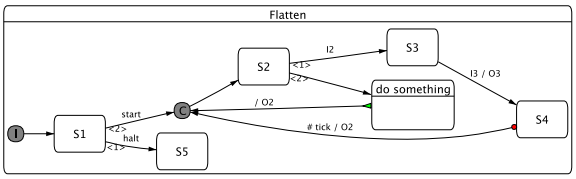
3. Optimization

As we have seen, the transformation produces verbose SSMs, but they have a very regular structure. This makes it possible to obtain a readable chart by applying some simple optimization rules. The rules are completely syntactical and make no assumptions on the actual execution of the SSMs, *e. g.*, whether transitions can actually be taken. Each rule takes one macrostate and transforms it. Neither information about a possibly surrounding macrostate nor about any substates is necessary. This is justified by the fact that we can replace any macrostate by another one with the same observable behavior.

In the following, we assume that only states which can actually terminate, *i. e.*, have at least one terminal state, have a normal termination transition originating from it. Similarly, a final state inside a macrostate without a normal termination is changed to a normal state. This constraint, imposed by Esterel Studio, assures that final states and normal terminations always match, which makes the chart easier to read. We also assume that states without an incoming transition, which therefore are unreachable, are removed. These conditions can easily be checked.



(a) A Statechart with a superfluous macrostate



(b) The same Statechart, after flattening

Figure 6: Example for removal of macrostates

Flattening Hierarchy

Since the state hierarchy is increased for every nesting of Esterel statements, the generated SSM contains usually much more hierarchy levels than necessary. In fact, one could also apply traditional Statechart flattening to remove *all* hierarchy levels; however, this can result in exponential state explosion. What we want to do is to remove hierarchy levels (macrostates) if this actually reduces the overall number of states and transitions. There are two cases where we may remove macrostates.

The first case is that a macrostate cannot be preempted, does not declare any local signals, variables, or history, and is not a parallel state. This may for example be generated for sequences (Transformation 6). Such a state can be removed and be replaced by its internal state transition graph. The initial state is replaced by a conditional pseudo-state, which connects all incoming transitions.

If no normal termination exists, we have to assure that the termination conditions do not change. This is done by changing the terminal attribute of inner states, depending on whether the macrostate we intend to remove is itself a terminal state.

If a normal termination exists, every contained final simple state gets an abortion which leads to the target of this normal termination, with the trigger immediate tick, where tick is a signal which is present in each instant. Thus these transitions are always enabled. These states, as the conditional pseudo-states that replace the initial state, might be removed by further optimization steps. Removing these states immediately would require further information about the surrounding state, leading to more complex rules. For each final macrostate, a normal termination is added, which leads to the same state and has the same effect as the normal termination of the surrounding state.

An example for the application of this rule can be seen in Figure 6. The state Superfluous is removed, and its initial state is replaced by a conditional node to which the transition from state S1 leads now. The self loop is replaced by a strong abortion from S4 and a normal termination from the macrostate do something to the same conditional. Neither S4 nor the do something state are final anymore.

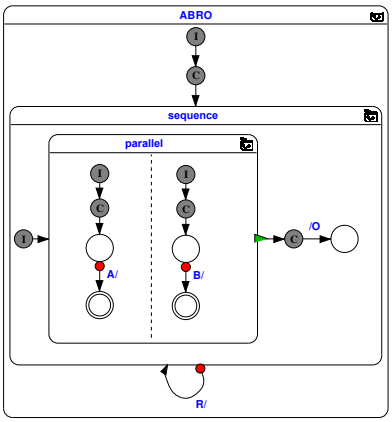


Figure 7: SSM for ABRO after unoptimized transformation (Figure 4) and subsequent flattening the loopeach, await, and emit states

The second case is that a macrostate contains only the initial state and one further state without any transitions except the initial one, as for example generated for emit statements (Transformation rule 2). Such a state can simply be removed. Again, the initial state is replaced by a conditional state. Local declarations of signals and variables are assigned to the surrounding state. Normal terminations are handled as in the first case, while other outgoing transitions are simply redirected and get the contained state as new source.

Applying these rules on the loopeach, await and emit states of the SSM for ABRO from Figure 4 yields the SSM in Figure 7, which already has a readable form, but is not optimal yet.

Removing Simple States

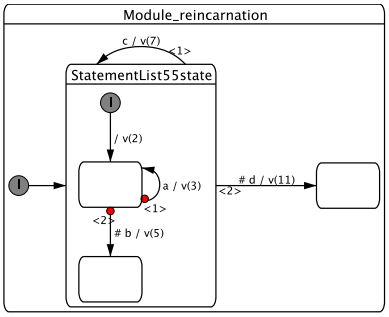
When removing macrostates, no simple states are eliminated: the initial state is transformed into a conditional state, and simple final states, which are only needed to indicate termination, are changed to transient, normal states. Both kind of states might be superfluous. We here consider the following cases.

- If a conditional pseudo-state has just one outgoing transition, it can simply be removed. Such states are inserted into the chart when macrostates that only have one initial-transitions are removed. We can use this rule to remove the conditional states from the ABRO SSM in Figure 7, which in this example suffices to produce an optimal chart.
- We can also remove simple states that have no internal actions and only one outgoing transition with trigger immediate tick and no condition. The incoming transitions are then redirected to the target of the unique outgoing transition.
- If a simple state has only one incoming and one outgoing transition that both have the same trigger and condition, it can be removed and the transitions be combined to one. This rule can only be applied if both transitions are delayed. Hence the incoming transition may not originate at an initial or conditional pseudo-state, since such transitions are always immediate. Furthermore, the state where the transitions originates must be a simple state without any internal actions.
- In SSMs, simple final states may neither have outgoing transitions, nor any internal actions. Thus, they only indicate the termination of the state and do not specify any further behavior. Therefore, all simple final states of a macrostate are interchangeable and can be replaced by one.

```
module reincarnation:
input a, b, c, d;
output v := 1 : combine integer
          with *;

loop
  weak abort
    emit v(2);
    loop
      await
      case a do
        emit v(3)
      case immediate b do
        emit v(5);
        halt
      end await
    end loop
  when
  case c do
    emit v(7)
  case immediate d do
    emit v(11);
    halt
  end weak abort
end loop
end module
```

(a) Esterel source



(b) Synthesized SSM

Figure 8: Transformation of the reincarnation example [3]

The iterative application of these simple rules on the generated charts produces in most cases well readable, relatively small charts. For the ABRO example from Figure 1a, the automatically synthesized SSM is, after optimization, identical to the SSM shown in Figure 2a, except for the naming of states. As another example consider Figure 8, where the Esterel code generated for the reincarnation example from André [3] is transformed back to its original, terse SSM.

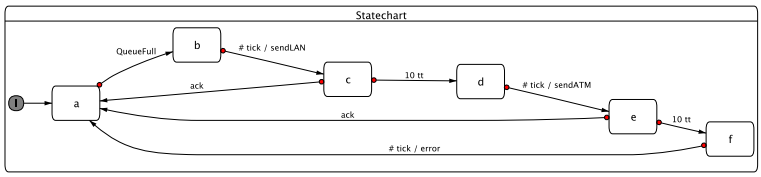
Note that even though these rules are motivated by the transformation from Esterel, they are general enough to be useful for SSMs in general. Unnecessary hierarchy and superfluous simple states can also be found in manually created charts. Especially novices tend to produce unnecessary large models with needless states, for example by splitting trigger and effect into separate transitions. An application of an optimization rule to a "real" SSM, which was not generated from Esterel but modeled by hand, can be seen in Figure 9. In fact, in the original model, the modeler did not introduce an explicit immediate tick, even though he intended the states to be transient.

4. Correctness of the Transformation

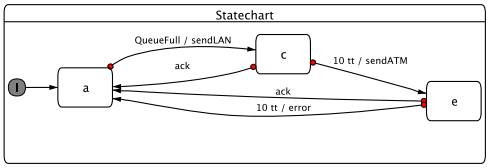
Both the transformation and the optimization were tested with various Esterel programs. While this gave good confidence in the transformation process, it does not prove the correctness in general, due to the possible complex interaction between the sub-statements of an Esterel statement. It has to be shown that the behavior of the generated SSM is equivalent to the behavior of the original Esterel program. It follows a sketch of an informal correctness proof. First, let us exclude traps from our considerations.

Esterel without Traps

For the Esterel kernel language, the control flow can only rest at explicit pause statements. Hence a configuration of an Esterel program is a set of currently active pause statements, called *registers* in this context. For the Esterel kernel language without traps, the behavioral equivalence can be proven by structural induction over all Esterel programs, giving a bi-simulation between the active registers of the Esterel program and the active states of a stable configuration of the constructed macrostate, *i. e.*, a configuration which can occur at the end of an instant.



(a) A Statechart with unnecessary, transient states



(b) Statechart after removing the transient states b, d, and f

Figure 9: Example for removing transient states

Since both Esterel programs and SSMs are fully deterministic, both can be executed in lock-step. For each register of the Esterel program, exactly one non-final simple state is generated. Whenever this state is active, the corresponding register of the Esterel program is also active. When a simple final state is active, the Esterel statement that corresponds to the macrostate has terminated, hence no register in it is active anymore. Most non-kernel statement preserve this correspondence. The expansion of halt, sustain, and await contain pauses. Accordingly their transformation into SSMs produces exactly one non-terminal state.

The loop each and every statements contain registers which are not directly expressed in the SSM. Instead of waiting in an explicit simple state, the termination of the included macrostate is not caught by a normal termination. For assessing correctness, both can be expressed by equivalent SSMs, which contain explicit states that are entered after the termination of the substate. These states correspond exactly to the registers of the Esterel program.

A little more involved is the behavior of suspend. The usual kernel statement suspend p when S executes p in the first instant regardless of the status of S. Therefore, a register inside the suspend statement is active whenever the suspend statement itself is active. This does not affect the bi-simulation. The statement suspend p when immediate S corresponds to await immediate not S; suspend p when S. The await contains an extra register which might be active even if p is not active. In the SSM, both behaviors can be modeled by a suspend transition of the macrostate. If this transition is tagged immediate, this might lead to a macrostate which is active, even though no substate in it is active. There is no simple state that corresponds to the new pause register. For the correctness proof, we would generate different SSMs for immediate and delayed suspension, where the immediate suspension contains an extra simple state, which waits for the absence of the trigger signal, before it starts the substate. The behavior of this macrostate is equivalent to the one we actually generate in the transformation.

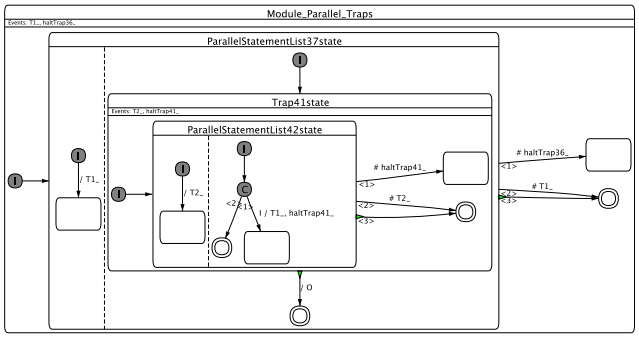
Traps

Raising an exception stops the execution of the control-flow of the current thread; however, all concurrent threads finish their execution for the current instant. This makes it possible that multiple, different traps are raised in the same instant. When multiple traps are raised inside a trap scope, only the handler of outermost trap is executed. If, however, a trap is raised in parallel to another trap-declaration, both handlers may be executed in the same instant.

Expressing traps in SSMs is similar to expressing traps by local signals and weak abort in the Esterel program, which is not trivial for the general case. This situation also arises, for example, in the Esterel derivate Quartz [21], which does not support traps directly, but weak and strong abortion. In contrast to signals, for exceptions it is important where they are raised, see also Figure 10a. Even though in this example the exception T1 is active in any case, the second exit T1, which is guarded by I, determines the output of

```
module Parallel_Traps:
input I;
output O;
trap T1 in
    exit T1
  ||
    trap T2 in
      exit T2 || present I then exit T1 end
    end trap;
  emit O
end trap
end module
```

(a) Esterel version



(b) SSM version

Figure 10: Example for transformation of nested traps into an SSM, illustrating the problems of replacing traps by signals

signal O. Therefore, it is in general not possible to replace each trap by only one local signal.

First, let us consider how this problem is handled in the context of Quartz. Since it is possible in Quartz to test which pause statements are active, preconditions on each raising of a trap can be computed. These preconditions can be used in an abort statement that replaces the trap, to test whether traps with higher priority were raised inside. Thus, the trap determines which exits are relevant for it. We use a different approach, which assigns extra signals to trap that indicate that their handler may not be executed, because a trap with higher priority was raised inside. The information for which traps the halt signal must be emitted is assigned to each exit statement.

We have also to show that the combination of macrostates derived from trap and exit statements behave correctly. First we notice that the macrostate derived from exit does not terminate. This assures that no statements later in a sequence are executed, while statements in parallel branches are executed normally. In the Esterel semantics, the raising of exceptions is usually encoded by their depth, *i. e.*, the number of trap declarations between the point where an exception is raised, and its declaration. When an exception is raised, this depth ($+2$, since 0 and 1 are used to encode nor-

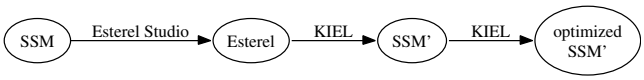


Figure 11: End-to-end validation of the transformation: From Esterel Studio SSMs to Esterel to *KIEL* SSMs

mal termination and pause) is returned as completion code. This code is passed to the surrounding statements until it reaches a trap. Here the completion code is examined. If it is greater than 2, which indicates that it is not the corresponding trap, it is decreased. If it has reached the corresponding trap, this value has reached 2; the handler of this trap is executed. This scheme is exactly preserved by our transformation, which sends a halt signal for all traps between the exit statement and the corresponding trap statement. In the SSM example shown in Figure 10b, the traphalt41_ signal determines whether the macrostate for the inner trap terminates and the O is emitted.

This transformation at the Esterel level can be easily extended to compound statements. Therefore, also for weak abortion a halt signal is needed. This halt signal is only emitted from inside an abortion. Observe that the exit that is used to implement the trap has always depth 2, *i. e.*, no traphalt signal needs to be emitted by a weak abortion.

Since a strong abortion suppresses the execution of its substatements in an instant where the abortion trigger is active, it can neither emit a halt signal, nor terminate in such an instant. Therefore, no halt signal is needed for strong abortion or statements like every and loop each, which are based on it.

Correctness of Optimizations

It remains to show that the behavior of the SSM is not changed by the optimizations. Since the optimizations consist of iteratively applied single rules, it suffices to argue that applying one rule does not change the behavior of a chart.

Obviously, the removal of superfluous normal terminations, unreachable states, and the conversion of final states into normal states, when no corresponding normal termination exists, does not change the behavior. The only crucial part in omitting hierarchy is that no signal declaration may be moved outside a self-loop. This is due to so called *schizophrenia* [5]; each activation of a macrostate in one instant creates new local signals. Since we prohibit explicitly the removal of macrostates that contain outgoing transitions and local signals, the behavior of schizophrenic signals is preserved.

The merging of final simple states is justified by the restriction on SSMs that a final state may neither have any on exit or during action nor outgoing transitions. Therefore, different final simple states are indistinguishable. Similarly, two simple states with the same outgoing transitions are merged only when the two states are indistinguishable. Then the correctness follows directly from the semantics of count-delayed transitions. The correctness of the removal of pseudo-states and simple states with trigger immediate tick and without a condition is straightforward.

Experimental validation

In addition to the theoretical considerations above, the transformation and optimization process and its implementation can be validated experimentally applying a round-trip tool chain that employs Esterel Studio's Esterel synthesis capabilities, see Figure 11. Starting with SSMs developed with Esterel Studio, one can employ Esterel Studio's Esterel code generator. From the resulting Esterel code, our transformation generates an equivalent SSM. After optimization, the round-trip produces SSMs corresponding to those starting the round-trip. When starting from well-written SSMs, they tend to be identical.

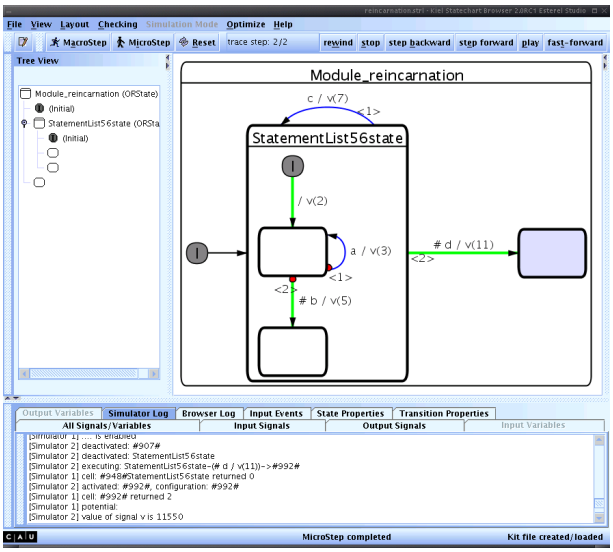


Figure 12: Screen-shot of *KIEL*—simulating the SSM synthesized from the Esterel module reincarnation (see Figure 8)

Similarly, one may perform a round-trip synthesis at the Esterel level, synthesizing a given Esterel program into an SSM with Kiel Integrated Environment for Layout (*KIEL*), and then synthesizing the same SSM into Esterel using Esterel Studio. We did this for all of the transformation rules individually, using dummy expressions for sub-statements. Due to Esterel Studio's rather elaborate Esterel synthesis the resulting Esterel programs were not identical to the original programs (*e. g.*, they contained a lot of spurious nothing statements), but they could relatively easily be proven to be equivalent [17] using the Structural Operations Semantics rules of Esterel's constructive behavioral semantics [5].

5. Implementation in *KIEL*

The *KIEL* tool is a prototypical modeling environment that has been developed to explore novel editing, browsing and simulation paradigms in the design of complex reactive systems [19]. A central enabling capability of *KIEL* is the automatic layout of Statecharts, which computes bottom-up layouts at each hierarchy level using GraphViz [12]. *KIEL* employs a generic concept of Statecharts, which can be adapted to specific notations and semantics, and it can import Statecharts that were created using other modeling tools. The currently supported dialects are those from Esterel Studio and from Matlab Simulink/Stateflow [22]. *KIEL* also provides an editor to create Statecharts from scratch or to modify imported Statecharts, and it provides a simulator. *KIEL* simulates the behavior of imported Statecharts according to the semantics of their original modeling tool. The automatic layout is also used to present different Statechart views for each Statechart configuration. Based on these views *KIEL* animates the simulation with a dynamic focus-and-context representation. *KIEL*'s built-in graphical editor also harnesses the layout capabilities for accelerated editing, for editing speeds close to textual editing. Figure 12 presents a screen-shot of *KIEL* as it simulates an SSM that has been synthesized (including automatic layout) from an Esterel module. *KIEL* is implemented in Java and is based on the *Model View Controller* (MVC) concept.

In *KIEL* we implemented the SSM synthesis and optimization described in Sections 2 and 3. As an alternative to import a Statechart from another tool or to editing a Statechart using the built-

Table 1: Experimental Results of SSM Synthesis

Model	Esterel	Safe State Machines									Time	
		Before Optimization				After Optimization						
	Lines of Code (LoC)	States/Pseudo-states	Transitions	Total Graphical Elements (GEs)	GEs/LoC	States/Pseudo-states	Transitions	Total Graphical Elements (GEs)	GEs/LoC	SSM Reduction Factor	Transformation [ms]	Optimization [ms]
ABRO	7	12/8	12	32	4.57	8/4	8	20	2.86	0.63	861	56
SCHIZOPHRENIA	10	13/9	14	36	3.60	4/3	6	13	1.30	0.36	838	81
REINCARNATION	25	27/17	28	72	2.88	5/2	6	13	0.52	0.18	642	83
JACKY1	27	31/19	33	83	3.07	11/5	12	28	1.04	0.34	913	93
RUNNER	55	39/24	42	105	1.91	21/11	25	57	1.04	0.54	725	160
TOKENRING3	79	77/61	91	229	2.90	15/20	38	73	0.92	0.32	733	278
GREYCOUNTER	82	211/148	254	613	7.48	42/50	106	198	2.41	0.32	789	593
ABCD	101	231/130	250	611	6.05	78/41	97	216	2.14	0.35	943	731
TOKENRING10	247	245/194	294	733	2.97	43/62	122	227	0.92	0.31	1267	736
MEJIA	555	374/246	414	1034	1.86	127/76	181	384	0.69	0.37	3085	1266
TCINT	687	475/285	543	1303	1.90	163/81	221	465	0.68	0.36	3382	1310
ATDS-100	948	961/558	1092	2611	2.75	352/184	504	1040	1.10	0.40	7046	2760
WW	1088	342/228	386	965	0.89	102/85	177	364	0.33	0.38	4470	1053
TOKENRING50	1207	1205/954	1454	3613	2.99	203/302	602	1107	0.92	0.31	6608	8148
TOKENRING100	2407	2405/1904	2904	7213	3.00	403/602	1202	2207	0.92	0.31	20910	25528
MCA200	7269	5159/3931	5947	15037	2.07	179/925	1794	2898	0.40	0.19	61510	100594

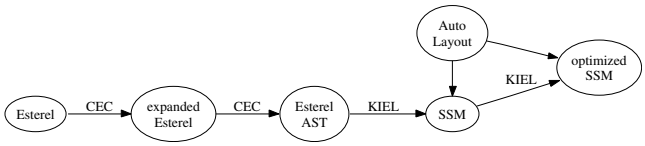


Figure 13: Tool chain transforming Esterel

in graphical editor, the modeler may simply import an Esterel v5 program, which is transformed on the fly into a SSM and can then also be saved as SSM. Figure 13 shows the tool chain used in *KIEL*. When importing an Esterel program, *KIEL* first employs the Columbia Esterel Compiler (*CEC*) [10] to perform the expansion of sub-modules. Then the *CEC* is used again to transform the code into an Extensible Markup Language (XML) formatted Abstract Syntax Tree (AST) representation. We process this using Java XML extensions and apply the transformation rules of Section 2 to produce the topology of the synthesized SSM. The *KIEL* auto-layouter augments this topology with graphical layout information, with numerous configuration options regarding the appearance of the synthesized SSM. The whole process is in general not noticeably slower than importing an already existing Statechart from another tool. As a final step, we may apply the optimization rules specified in Section 3. This is typically done all at once, but if the user wishes to trace the effect of individual optimization rules, one may also perform the optimizations step-by-step. After each step, a new SSM layout is computed.

To assess the efficacy and efficiency of *KIEL*, Table 1 presents experimental results for transforming various Esterel benchmarks, most of them are taken from the Estbench Esterel benchmark suite [9] and the *CEC* distribution [10]. The table compares the size of Esterel code with the complexity of the corresponding synthesized SSMs. *Lines of Code* denote the overall size of the (module expanded) Esterel code. The *Graphical Elements* category characterizes the graphical complexity of the resulting SSM models before and after applying the optimization. We observe that as the degree of complexity of the resulting chart increases, the ratio between these increases as well, which is desirable to minimize the number of graphical elements. The element-wise reduced Statechart is more compact and has less overhead and redundancy; the optimization often reduces the Statechart by a factor of three or more. Hence, in our experience a synthesized, optimized Statechart generally is very readable and comprehensible. As an example of intermediate size, Figure 14 shows the component "mode selection" of the canonical wrist watch example [9, 15]. This component contains 38 states, 29 pseudo-states and 62 transitions, hence a total of 129 graphical elements.

To quantify efficiency, the *Transformation* times indicate how long it takes to load, expand, parse and transform an Esterel program. The computation times were measured on a PC with Linux OS, an 2.6 GHz AMD Athlon 64 processor and 2 GB of RAM. For large models, this time becomes noticeable, but typically takes at most a couple seconds. Similarly, the time to compute the optimized SSM version is less than three seconds for most of the benchmarks considered here. Only the optimization of the industrially sized example MCA200 takes about 100 seconds.

6. Conclusions and Further Work

Embedded devices are proliferating, and their complexity is ever increasing. Statecharts are a well established formalism for the description of the reactive behavior of such devices. However, there is evidence that the current use of this formalism is reaching its limits for development. The larger a model description becomes, the less traceable and manageable it gets. A well-established alternative for specifying reactive systems are textual approaches. To benefit from the general ease of handling textual programs as well as from the intuitiveness of graphical languages, it can be useful to transform one to another. Another motivation for doing so could be the desire to separate structure and layout—akin to, for example, the use

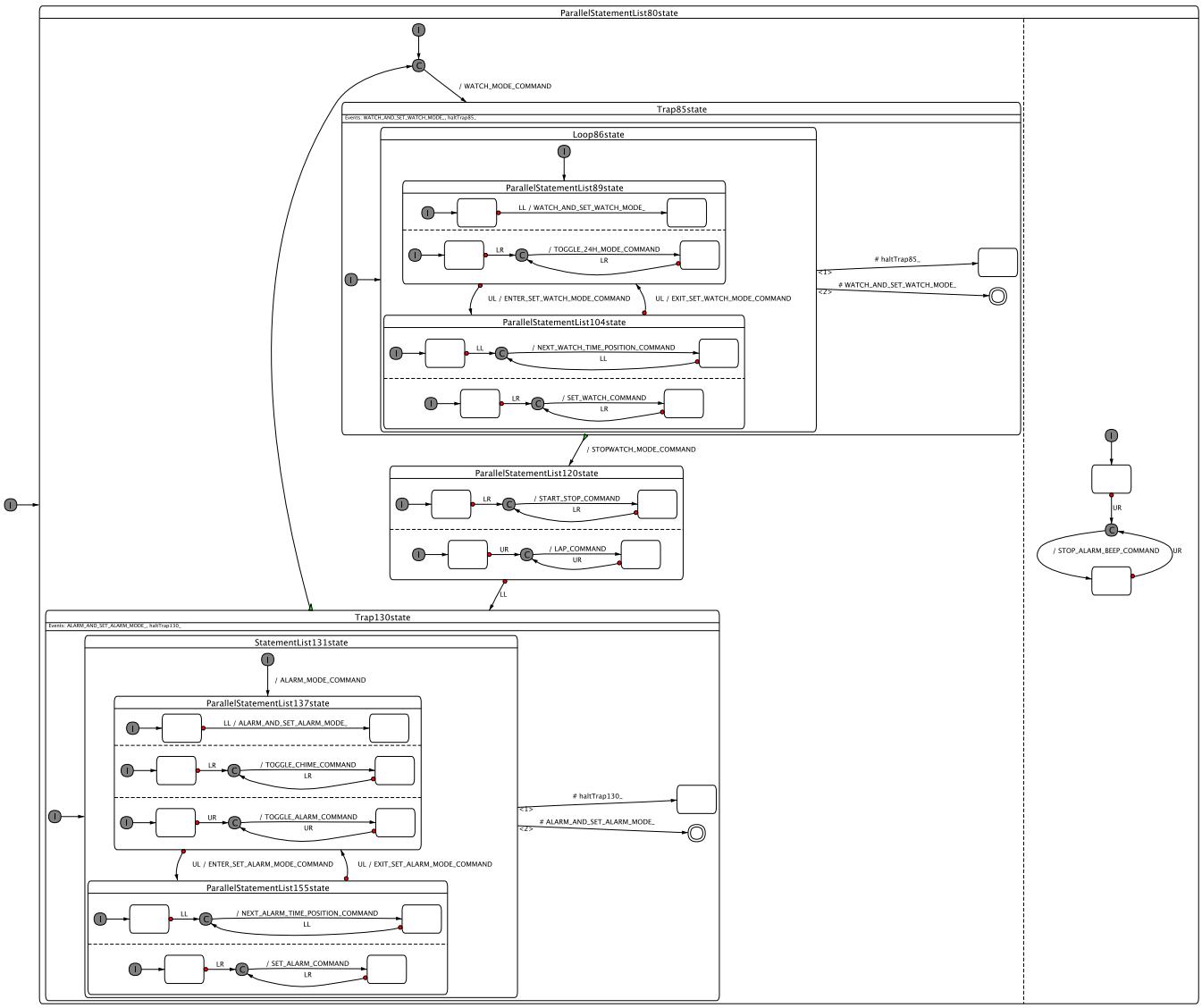


Figure 14: The component “mode selection” of the wrist watch example

of a document processing system (such as LaTeX) that produces a nicely typeset document that adheres to certain formatting guidelines from an ASCII source. Such a system lets the author focus on the contents of the document without worrying too much about the layout.

We have presented such a transformation for the case of SSM dialect of Statecharts, using Esterel as input language. The transformation consists of derivation rules, whose application to Esterel statements performs the successive synthesis of SSMs. The transformation is accompanied by optimizations, which are applicable after the initial transformation, to reduce the complexity of SSMs. While different Statechart dialects employ different execution semantics and the synthesis of Statecharts must respect the subtleties of the semantics, we expect that the transformation and optimizations presented here could be adapted to Statechart dialects other than SSMs.

We have experimentally validated the transformation and optimizations and have argued their correctness here, albeit informally. We have implemented the transformation in *KIEL*, and preliminary experience with this tool indicates the practicality of the approach. A central enabling capability is the automated layout of Statecharts to place the synthesized objects. Experimental results are very promising regarding the transformation efficiency. Especially the automatic layout of Statecharts is a promising basis for similar work synthesizing and displaying Statecharts.

We intend to improve our optimization process and to add further rules, for example to remove redundant signals due to the transformation of the Esterel `trap` statements. Note, however, that it is not always obvious what really constitutes an optimization. For example, one might argue that macrostates should be preserved if they have an outgoing transition and multiple final states, as was for example the case in the SSM shown in Figure 6; in this example each final state results in an additional transition, and the Write Things Once principle might be violated for the transition labels. Hence in general it is desirable to allow the selective enabling of individual optimization rules.

Related to the field of optimizing Statecharts is the definition of *design guidelines* regarding “good” Statecharts, and the definition

of *robustness checks* that indicate likely modeling errors, similar to lint in the context of C programming. Consider again the example in Figure 9, where the modeler had (needlessly) split transition triggers and actions and (accidentally) introduced delays. The absence of a trigger, and hence the implicit waiting for the next instant, is seldom really wanted by a modeler. A syntactical checker might produce a warning for those states and advise the modeler to insert explicit immediate tick triggers—which would then permit the subsequent merging of transitions. We are currently investigating such robustness checks, which we consider to be a valuable augmentation of the optimizations presented here.

Finally, we plan to explore augmentations to the transformation presented here that support the modeling of complex systems. Statecharts allow to reduce the size of individual charts by "collapsing" a macrostate in the chart where it is referenced, and describing the internal structure of the collapsed macrostate on a separate sheet. It has been our experience with the transformation presented here that the compactness of the synthesized Statecharts and the nature of the resulting design flow—the user does not have to edit the Statechart itself—allow to present more detail on a single sheet than is possible with Statecharts created manually. For example, we feel that the "mode selection" Statechart presented in Figure 14 is still fairly readable; however, one probably would not have created such a Statechart manually, simply because graphical editing of Statecharts of this size is rather tedious. Nonetheless, for very complex models it might still be desirable to collapse macrostates to keep the individual Statechart readable. For example, one probably would not want to inspect the complete wrist watch in full detail at once (unless one has a very large screen or prints on large paper). For visualizing complex models during simulation, the *KIEL* tool already supports a dynamic focus-and-context representation that automatically collapses inactive macrostates [19]. Alternatively, it would be possible to enrich the Statechart synthesis procedure with customizable rules on when macrostates should be expanded and when they should be collapsed.

Acknowledgments

Lars Kühl has performed the implementation of the transformation presented here, Mirko Wischer and the rest of the *KIEL* development team have also contributed. Charles André has been a valuable discussion partner on this subject and has suggested improvements to earlier versions of this paper. We also thank the Jan Lukoschus, Gunnar Schaefer, Jan Täubrich and the anonymous reviewers for their helpful comments.

References

[1] ANDRÉ, C. Representation and Analysis of Reactive Behaviors: A Synchronous Approach. In *Computational Engineering in Systems Applications (CESA)* (Lille (F), July 1996), IEEE-SMC, pp. 19–29.

[2] ANDRÉ, C. SyncCharts: A Visual Representation of Reactive Behaviors. Tech. Rep. RR 95–52, rev. RR (96–56), I3S, Sophia-Antipolis, France, Rev. April 1996.

[3] ANDRÉ, C. Semantics of S.S.M (Safe State Machine). Tech. rep., Esterel Technologies, Sophia-Antipolis, France, Apr. 2003. available at `http://www.esterel-technologies.com`, in the download section.

[4] ANDRÉ, C. Computing SyncCharts reactions. *Electronic Notes in Theoretical Computer Science 88* (Oct. 2004), 3–19.

[5] BERRY, G. *The Constructive Semantics of Pure Esterel*. Draft Book, 1999.

[6] BERRY, G. *The Esterel v5 Language Primer, Version v5_91*. Centre de Mathématiques Appliquées Ecole des Mines and INRIA, 06565 Sophia-Antipolis, 2000.

[7] BERRY, G., AND GONTHIER, G. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming 19*, 2 (1992), 87–152.

[8] CASTELLÓ, R., MILI, R., AND TOLLIS, I. G. A Framework for the Static and Interactive Visualization for Statecharts. *Journal of Graph Algorithms and Applications 6*, 3 (2002), 313–351.

[9] Estbench Esterel Benchmark Suite. `http://www1.cs.columbia.edu/~sedwards/software/estbench-1.0.tar.gz`.

[10] EDWARDS, S. A. CEC: The Columbia Esterel Compiler. `http://www1.cs.columbia.edu/~sedwards/cec/`.

[11] FLORENTZ, B., MUTZ, M., AND HUHN, M. Avoiding unpredicted behaviour of large scale embedded systems by design and application of modelling rules. In *Proceedings of the 2004 First International Workshop on Model, Design and Validation* (Nov. 2004).

[12] GANSNER, E. R., AND NORTH, S. C. An open graph visualization system and its applications to software engineering. *Software—Practice and Experience 30*, 11 (2000), 1203–1234.

[13] GUERNIC, P. L., GOUTIER, T., BORGNE, M. L., AND MAIRE, C. L. Programming real time applications with SIGNAL. *Proceedings of the IEEE 79*, 9 (Sept. 1991).

[14] HALBWACHS, N., CASPI, P., RAYMOND, P., AND PILAUD, D. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE 79*, 9 (September 1991), 1305–1320.

[15] HAREL, D., LACHOVER, H., NAAMAD, A., PNUELI, A., POLITI, M., SHERMAN, R., SHTULL-TRAURING, A., AND TRAKHTENBROT, M. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering 16*, 4 (Apr. 1990), 403–414.

[16] JAFFE, M. S., LEVESON, N. G., HEIMDAHL, M. P. E., AND MELHART, B. E. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering 17*, 3 (Mar. 1991), 241–258.

[17] KÜHL, L. Transformation von Esterel nach Esterel Studio. Diplomarbeit, Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik, Sept. 2005.

[18] OBJECT MANAGEMENT GROUP. Unified Modeling Lanugage—UML Resource Page. `http://www.uml.org`.

[19] PROCHNOW, S., AND HANXLEDEN, R. V. Comfortable Modeling of Complex Reactive Systems. In *Design, Automation and Test in Europe 2006 (DATE'06)* (München, March 2006).

[20] RATIONAL SOFTWARE. Rational rose technical developer. `http://www-306.ibm.com/software/awdtools/developer/technical/`.

[21] SCHNEIDER, K. Embedding imperative synchronous languages in interactive theorem provers. In *Conference on Application of Concurrency to System Design (ACSD)* (Newcastle upon Tyne, UK, June 2001), IEEE Computer Society, pp. 143–156.

[22] THE MATHWORKS. Stateflow—Design and simulate event-driven systems. `http://www.mathworks.com/products/stateflow/`.