Translating SHIM to Guarded Actions

Jens Brandt[1], Klaus Schneider[1], and Stephen A. Edwards[2]

[1] Department of Computer Science, University of Kaiserslautern,
lastname@cs.uni-kl.de
[2] Department of Computer Science, Columbia University, New York,
sedwards@cs.columbia.edu

Abstract. SHIM is a concurrent deterministic programming language for embedded systems built on rendezvous communication. It abstracts away many details to give the developer a high-level view that includes virtual shared variables, threads as orthogonal statements, and deterministic concurrent exceptions.
In this paper, we present a new way to compile a SHIM-like language into a set of asynchronous guarded actions, a well-established intermediate representation for concurrent systems. By doing so, we build a bridge to many other tools, including hardware synthesis and formal verification. We present our translation in detail, illustrate it through examples, and show how the result can be used by various other tools.

1 Introduction

Describing and programming concurrent systems challenges developers. The state space of such systems is generally large and developers can rarely imagine every situation a system might encounter. Compounding the problem, system models often employ low-level communication and synchronization primitives such as threads and locks expressed with a sequential programming language, where experience has shown that non-trivial programs are very hard to analyze. Extensive and expensive testing is commonly used to improve assurances of safety and liveness. Given that today's embedded systems are generally parallel, this problem is growing more pernicious because the correctness of such systems is often a matter of life and death.

Adopting a model of computation, a meta-model targeted at a class of applications that defines general rules for communication and computation, is one may to deal with the problem. Employing such a model restricts the complexity of a system and thus the difficulty of validating it. This raises the level of abstraction, providing both the programmer and automated tools an easier way to gain insight into a system.

Edwards and Tardieu's SHIM language [7] is based on such a model of computation targeted at the domain of embedded systems. It provides many features aimed at integrating hardware with software. Its imperative programming style makes it accessible to many developers, but it resolves many issues that usually arise in such a setting. Specifically, its restriction to rendezvous-style communication ensures parallel execution remains deterministic, i.e., so

that repeated runs of a program with the same input always produce the same output. Furthermore, its state space remains finite to enable simpler hardware synthesis [16]. It hides communication behind virtual shared variables, presenting developers with a simple high-level view. Additionally, it has direct support for running statements (and hence threads) in parallel and support for concurrent exceptions that remain deterministic.

Creating the compiler infrastructure for new programming languages is a difficult, time-consuming task that requires experts in the source language, program analysis, and target architectures. A classical solution is to use a common intermediate format that bridges the gap between powerful programming languages with complex semantics and the low-level description of the target code. Such an intermediate representation also allows designers to share common components of the compiler infrastructure: new input languages can be added by simply implementing the front-end; additional back-ends support new target architectures.

A common solution to divide these tasks is to define an intermediate representation that is independent of the target architecture on the one side but does not contain the complexity of the source language on the other side. For traditional sequential programming languages, control-/data-flow graphs are such a model. As they are generally bound to the program structure, they are rarely suitable for concurrent models, although there are some exceptions.

In this paper, we propose the use of asynchronous guarded actions as an intermediate format. They are a well-established concept for the description of concurrent systems. With a theoretical background in term rewriting systems, they have been used in many specification and verification formalisms (e.g., Dijkstra's guarded commands [5], Unity [3], Murφ [6], DisCo [12]) and they have also shown their power in hardware and software synthesis (e.g., Concurrent Action-Oriented Specifications [11]).

Our core technical contribution is a translation algorithm that takes a SHIM-like program and generates asynchronous guarded actions for it. By translating SHIM to guarded actions, we leverage many existing techniques for hardware/software codesign and formal verification. The approach presented in this paper complements existing compilation techniques for SHIM, which focus on software generation. Furthermore, by using the conflict analysis presented by Brandt et al. [2], the SHIM approach can harness all the implementation and verification techniques developed for the synchronous world.

As mentioned above, related work focused mainly on software generation. Edwards and Tardieu introduced initial ideas [7,15,14] and later refined them. For example, Edwards and Tardieu [8] showed how threads can be statically scheduled to minimize the synchronization overhead in generated code. Later, Edwards, Vasudevan, and Tardieu [9] use Pthreads as a target for software synthesis. Edwards and Tardieu [7] sketch a rudimentary hardware generation, but only cover a subset of the language. While deadlock detection [17] and buffer sharing [18] has been considered for SHIM, much work remains to be done on formal verification and static analysis.

This paper is structured as follows: In Section 2, we present our source language—a SHIM dialect called Emerald. Then, in Section 3, we describe our intermediate representation. We describe the translation between these two models in Section 4. In Section 5, we show how using our translation enables the use of existing hardware synthesis and symbolic model checking tools. We conclude in Section 6.

2 The Description Language Emerald

2.1 Overview

We translate Emerald, a clone of the SHIM language [15]. As in SHIM, system descriptions are a set of concurrently running processes that only exchange data by rendezvous-style communication. Emerald supports the same set of orthogonal statements, deterministic best-effort exceptions and virtual shared variables to hide lower-level communication.

To a SHIM base, Emerald adds support for built-in assertion-style verification and different kinds of preemption. Emerald's type system includes fixed bit-width integers with a complete set of arithmetic operations, which is the foundation for hardware synthesis and formal verification. All of this was borrowed from the synchronous programming language Quartz[13], enabling developers to integrate components written in either languages to only consider temporal aspects. Such borrowing also enabled us to reuse infrastructure from the Averest system, which provides support at the front-end for the general program structure, expressions and type checking and synthesis tools at the backend, which will be used for the experiments in Section 5. However, the techniques in this paper do not rely on these additional features; our compilation technique can be applied directly to SHIM programs.

Below, we give a brief overview of the core of the Emerald language. For lack of space, we do not describe its semantics in detail; instead see Edwards and Tardieu [7,15]. The Emerald core consists of the statements listed below, where S, S_1, and S_2 are also core statements, x is a variable, e an exception, σ is a Boolean expression, and α is a type.

`nothing`	Empty statement
`halt`	Stop this thread
$x = \tau$	Assignment
`pause`(x) / `pause`	Synchronization
`if` (σ) S_1 `else` S_2	Conditional
$S_1; S_2$	Sequence
`do` S `while`(σ)	Iteration
$S_1 \parallel \ldots \parallel S_n$	Concurrency
`throw` e	Throw exception
`try` S_1 `catch`$(e) S_2$	Catch exception
$\{\alpha\ x;\ S\}$	Declare x of type α in scope S

```
module Parallel02                module Parallel02:
(bool x) {                       init:
  int a = 0;                       True => a.1 = 0
  int b = 0;                       True => b.1 = 0
  int c = 0;                       True => c.1 = 0
                                   True => a@1 = a.1
  {                                True => b@1 = b.1
    a = 1;                         True => c@1 = c.1
    pause(a);                      True => next(pc'thr000) = 1
    a = 2;                         True => ec'thr001 = 0
  } || {                           True => a.2 = 1
    b = b+a;                       True => next(pc'thr001) = 1
    pause(a);                      True => ec'thr002 = 0
    b = b+2*a;                     True => b.2 = a@1+b.1
  } || {                           True => next(pc'thr002) = 1
    c = a+b+c;                     True => ec'thr003 = 0
    pause(a);                      True => c.2 = c.1+a@1+b@1
    c = a+2*b+3*c;                 True => next(pc'thr003) = 1
  }                                True => next(a) = a.2
                                   True => next(b) = b.2
}                                  True => next(c) = c.2

rules:
  a: when(
       pc'thr001==1 &
       pc'thr002==1 &
       pc'thr003==1)
     True =>
       a@1 = a
     pc'thr001==1 =>
       a.1 = 2
     pc'thr001==1 =>
       next(a) = a.1
     pc'thr002==1 =>
       b.1 = b+a@1*2
     pc'thr002==1 =>
       next(b) = b.1
     pc'thr003==1 =>
       c.1 = →
             a@1+b@1*2+c*3
     pc'thr003==1 =>
       next(c) = c.1
```

Fig. 1. Example `Parallel02`

`nothing` is the empty statement; `halt` stops, but does not terminate the thread in which it resides. The assignment $x = \tau$ evaluates the expression τ in the current environment and assigns its value to x. `pause`(x) forces the current thread and all others that know about the shared variable x to synchronize. Additionally, there is a `pause` statement without parameters, which corresponds to internal transitions in CSP [10]; we use it to denote clock cycle boundaries in hardware synthesis. Conditional, sequence and iteration work as in any sequential program. Concurrency is added by the statement $S_1 \parallel \ldots \parallel S_n$, which forks statements $S_1, \ldots, S_n$ as parallel threads. The parent thread is immediately suspended and resumes when all children have terminated. The execution of parallel threads is only constrained by synchronizations `pause`(x) on shared variables x as explained below. `try` S_1 `catch`(e) S_2 declares and exception e in scope S_1. If it is thrown by `throw` e, the statement tries to catch it and calls the handler S_2.

In the next section, we discuss in more detail the behavior of shared variables and exceptions—two important aspects of the language semantics upon which the compilation algorithm hinges.

2.2 Shared Variables and Synchronization

The underlying model of computation in Emerald is a restricted form of communicating sequential processes. However, communication is hidden in the

source language by virtual shared variables and synchronization on them. A variable can be declared at an arbitrary point in the program and accessed in several concurrently running threads. To avoid race conditions, which make the overall behavior generally nondeterministic, a variable x can be only written by a single thread. If a thread forks several subthreads, it hands over the write access to a single child. Only this thread has access to the current value of a variable x; all other threads can only read the value that the variable has had at the last common synchronization point. For concurrent threads accessing x, this is either the fork point or a point where all of them executed a $\texttt{pause}(x)$ simultaneously. Thereby, the order in which concurrent threads advance does not have an effect on the functional behavior of the program.

If a thread synchronizes on x by calling $\texttt{pause}(x)$, the underlying rendezvous-like communication requires all concurrently running threads reading or writing x to synchronize on x. The thread blocks and does not participate in any other communication until all others also reach a $\texttt{pause}(x)$ or terminate. Although the possibility of a thread synchronizing on several variables introduces the danger of deadlocks, it is the key to guaranteeing determinism. Note that due to this synchronization, the parallel operator $\parallel$ is not associative and therefore, the language core contains the general variant for creating n threads instead of simply two. Thus, $S_1 \parallel (S_2 \parallel S_3)$, $S_1 \parallel S_2 \parallel S_3$, and $(S_1 \parallel S_2) \parallel S_3$ may exhibit different behaviors because of different synchronization on shared variables.

The left column of Figure 1 illustrates this. After initializing variables a, b, and c, the program starts three threads that share all the variables. Since a is written by the first thread, the other two threads can only read it. These readers use $a = 0$ in their first assignments, since the update $a = 1$ is made accessible to them only by $\texttt{pause}(a)$ statement. Since b is never communicated, the last thread still uses $b = 0$ in its last assignment. Hence, the example terminates with $a = 2$, $b = 2$, and $c = 1$.

2.3 Best-Effort Exceptions

State-of-the-art sequential programming languages provide exceptions to deal with uncommon situations in a structured way. Once such an exception is thrown at an arbitrary point in the program, it can be caught and control can be handed over to dedicated handler, which is responsible for dealing with the situation. The benefits of exceptions in sequential programs are indisputable, so following the concept is also interesting in a parallel programming language such as Emerald. However, making them deterministic in a concurrent context is very challenging.

Edwards and Tardieu tackled the problem by so-called best-effort preemption [7], in which catching exceptions might fail. Their design decision leads to programs that may get stuck instead of producing a nondeterministic behavior, which is generally the better alternative for testing and verifying parallel programs.

```
module Exception02 (bool x) {
    int a; int b; int c;
    try {
            {
                throw e;     // thread exits here
                pause(a);
                a = 1;
            } || {
                b = 2 + a;
                pause(a);    // thread exits here
                b = 3 + a;
                pause(b);
            } || {
                c = 4 + b;
                pause(b);    // thread exits here
                c = 5 + b;
            }
    } catch(e) { a = b + c; }
}
```

Fig. 2. Example `Exception02`

Similar to the design of shared variables, the fact that an exception is thrown in a thread is not immediately propagated to other threads because there is no good definition for "immediately" in an asynchronous setting such as Emerald. One choice would be to notify threads at an undefined point, a source of non-deterministic behavior. Instead, exceptions in Emerald are only propagated to other threads communicating with the emitter of the exception. In this case, the other thread immediately becomes a repeater of that exception. Finally, a set of concurrent threads is only aborted if all concurrently running threads have terminated.

The example shown in Figure 2 illustrates deterministic preemption. The first thread throws the exception e and aborts immediately. The other threads proceed until they communicate with a "poisoned" thread. The second thread will abort at `pause`(a), which will cause the third thread to abort at `pause`(b). When all subthreads have exited, their parent will also exit so the exception handler can be called. Hence, $a = 6$ at the end.

3 Asynchronous Guarded Actions

Our translation process expresses Emerald programs as groups of asynchronous running guarded actions. They are defined over a set of explicit variables $\mathcal{V}$, which represent the state of the modeled component. The behavior is described by a set of *rules*, which are guarded atomic actions of the form `rule` r_i`when`(γ_i) B_i, where γ_i is the guard and B_i the body of rule r_i. The body of a rule B_i is a set of synchronous guarded actions of the form $\langle \gamma \Rightarrow x = \tau \rangle$ (for an immediate

assignment) or $\langle \gamma \Rightarrow \texttt{next}(x) = \tau \rangle$ (for a delayed assignment). Both kinds of assignments evaluate the right-hand side expression τ in the current macro step. Immediate assignments $x = \tau$ write the obtained value of τ immediately to the variable x (so that other actions see the effect), whereas delayed ones $\texttt{next}(x) = \tau$ write the value for the following macro step.

For the interaction with the environment, the target model makes use of so-called methods, which are parameterized rules of the form $\texttt{method}\ m(p_i 1, \ldots)\ \texttt{when}(\gamma_i)\ B_i$. In addition to the local variables, the actions of a method have access to the variables specified in its parameter list. These parameter lists may contain inputs and outputs. In the concrete syntax, we use the prefix `?` to mark a parameter as an input and `!` to mark a parameter as an output.

The semantics of the asynchronous guarded actions is rather simple. After the initialization of all variables, the following two steps are repeated forever: first, the guards of all actions are evaluated with respect to the current state. Among the actions whose guards evaluate to true, an arbitrary one is chosen and its body is executed. Inside the body, there are multiple synchronous actions, which are considered to execute in parallel. We consider this model as an asynchronous one because there is no notion of synchronous execution of multiple rules in the semantics. Rules execute one by one as their guards become true, in an arbitrary order. Since the execution generally modifies the system state, other actions will be possibly activated in the following iteration. If no action is activated, the loop may be also aborted, since no further state change will ever occur.

Let q_0 be the initial state of the system, and $q \xrightarrow{S} q'$ indicate that action S transforms the system in state q to state q'. Then, a run of a model is a sequence of system states $\langle q_0, q_1 \ldots \rangle$ where $q_i \xrightarrow{S_x} q_{i+1}$ and $\texttt{when}(\gamma_x)\ \mathcal{C}_x$ is an arbitrary action which is activated in state q_i, i.e., $q_i(\gamma_x) = \mathsf{true}$. Obviously, the system description is nondeterministic: even in the presence of the same inputs, which lead to the same activation of guards, the system can produce different outputs by choosing different activated actions. Hence, models consisting of asynchronous guarded actions are generally intended to be specifications, which describe a set of acceptable implementations.

4 Compilation

This section presents the core of our contribution: the translation of Emerald programs to asynchronous guarded actions. First, we give an overview of the translation process in the following section before we highlight some particular aspects, which we discussed in Section 2: synchronization on shared variables and best-effort exceptions.

4.1 Overview

We divide the translation of Emerald programs into several stages. After parsing the program, the abstract syntax tree is checked and enriched with the following information.

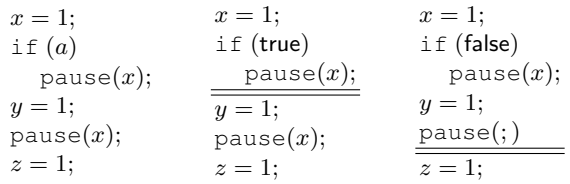

Fig. 3. Surface and Depth

- For each thread t, a fresh integer pc_t variable is created, which serves as its program counter. Furthermore, for each position where the control flow can rest (namely the synchronization $\texttt{pause}(x)$, the break-down `halt` and the parallel statement) integer labels are fixed for the program counter. This information is added as parameters to the corresponding program statements (e.g., $\ell_1 : \texttt{pause}(x)$), in the presentation below as well as in our actual implementation. Thereby, the label 0 always refers to the initial place, and label -1 indicates that a thread has exited due to an exception.
- For each thread t, an additional variable ec_t is created, which stores its current error code. For each exception e, a unique integer code is fixed, which will be stored in the error code of its thread if the exception is thrown. Otherwise, the error of of a thread is set to 0, which indicates that no exception has been thrown.

The actual compilation of the program is performed by two recursive functions, which are responsible for the *surface* and the *depth* of a statement. Intuitively, the surface consists of the actions that are executed when the statement is started, i.e., all the parts that are executed before reaching one of the next labels (attributed to synchronization, halt, and parallel statements). Thus, the surface refers to the part that can be executed within a single rule (recall that the body of a rule generally contains several actions). The depth contains the statements that are executed when the program resumes execution for the next rules, i.e., when the control is already inside the program and proceeds with its execution. It is important to see that surface and depth overlap, since labels may be attributed to conditionally executed statements. Consider the example in Figure 3: while the action $x = 1$ is only in the surface and the action $z = 1$ is only in the depth, the action $y = 1$ is both in the surface and depth of the sequence.

This compilation scheme [1] has the advantage that branches of the program can be compiled depending on the label from where they have been reached. As we will show below, this technique for Emerald compilation provides many benefits over a straightforward single recursive traversal over the program structure.

The surface compilation is implemented as a recursive traversal over the program structure, e.g., for a loop we first compile the loop body and then add the loop behavior. While descending in the recursion, the compilation algorithm passes down the following parameters:

- t is the ID of the current thread

- S is the statement to compile
- φ is the activation condition of S
- ρ is a substitution mapping program variables to variables in the intermediate code

Each call returns the following intermediate result:

- I is a Boolean condition describing when the statement is instantaneous (i.e., when the next label is not reached)
- $\mathcal{R}$ is the set of synchronous guarded actions extracted from the surface of the statement
- ρ is a substitution reflecting the current variable context after the execution of S

The *surface compilation* of the primitive statements is straightforward. `nothing` is always instantaneous and it does not produce any actions. In contrast, `halt` is not instantaneous and creates an action which moves the control flow to a dead-end position. The assignment $x = \tau$ is instantaneous. It updates ρ as described in Section 4.2 and creates an action that performs the assignment. The statement ℓ : `pause`(x) always concludes the surface and creates an action that moves the control flow to the synchronisation position ℓ (which has been generated in the preprocessing step). For all the actions generated by the primitive statements, the current start condition φ is used as the guard.

Throwing the exception e is compiled to a movement of the control flow to the exit position -1. The exit code of the current thread is updated if the currently thrown exception is of higher precedence than the previously stored one (for details see Section 4.4).

For the compilation of the sequence statement, the compile algorithm processes both parts of the sequence and concatenates their actions. The second statement is added under the condition that the first one is instantaneous. The compilation of the conditional is straightforward: its guard σ is added to the recursive calls of the compilation algorithm, and it is used for the combination of the partial results. The iteration statements follows the same principle and simply compiles its body. Thereby, it is assumed that the loop body is not instantaneous (i.e., that there is no statement with a label in the loop). If this were the case, this loop would immediately restart leading to an infinite empty loop with a potential lack of reactivity. Programs that contain such loops are considered to be ill-formed, and the compiler rejects them.

The surface of the parallel statement contains the actions for the fork, which initialize the program counter, exit code and local variable copies of the subthreads, in addition to the actual surfaces of its subthreads. Obviously, the whole parallel statement is instantaneous if all its subthreads are so.

Finally, the compilation of the *try-catch* statement and the compilation of the scope of a local variable simply compile the inner parts. For the local variable, an action initializing the variable to its default value is prepended to the compiled actions.

The *depth compilation* is implemented in a similar way as a recursive traversal over the program structure. Thereby, depth compilation makes use of the following parameters

- t is the ID of the current thread
- S is the statement to compile
- ρ is a substitution reflecting the current variable context (similar to the ρ used in the surface compilation)

and returns a structure containing the following information:

- A is a Boolean condition that holds when the statement is active
- T are Boolean conditions indicating when the current set of rules terminate
- E is a Boolean condition that holds when all subthreads have exited due to an exception
- $\mathcal{Y}$ stores for each shared variable the condition when a synchronization on them is requested by the program threads
- $\mathcal{R}$ is the current set of rules extracted from the statement

Since `nothing` does not have a depth part, it returns an empty result, where the set of rules is empty, and the A and T conditions are set to false. The same holds for the assignment $x = \tau$ and the `throw` statement. For ℓ : `halt`, we do not produce any rules either but set the T condition to its label ℓ.

The statement ℓ : `pause`(x) creates a new rule. Its position ℓ is also used as the synchronization condition for x in the map $\mathcal{Y}$. The same condition is used for the predicates A and T.

For the compilation of the sequence statement, the compile algorithm has to make three recursive calls, since the depth of the whole statement generally consists of the depth of the first part, the surface of the second part and the depth of the second part. The rules and synchronization maps of the depth parts are simply merged, and the actions from the surface call are appended to all open rules of the first statement (which is given by the T conditions). The compilation of the conditional simply compiles both branches and assembles the results. In the depth, the guard σ does not play any role, since it has been already checked in the surface. The iteration statements basically follows the same principle as the sequence. First, the depth of the body is compiled. Then the surface is appended to all open rules, with σ as an additional condition that holds for reentering the loop body. By splitting the compilation into surface and depth, the reenter condition for the loop can be computed before it is used so that acyclic descriptions are generated by construction. Similar to the surface compilation of the loop, we check that the loop body is not instantaneous if the loop is reentered.

The most challenging statement for the compilation algorithm is the depth of the parallel statement since it must combine all information about the subthreads and their interaction. First, the depth parts of all substatements are compiled and their compilation results are merged. The rules and activation conditions can be combined by a simple union and disjunction, respectively. In contrast, the termination condition is not just the conjunction but it must also consider that the subthreads terminate at different points of time. Therefore, termination happens when the last thread (all others are not active (A) anymore) terminates. Furthermore, new rules are generated that handle the propagation of exceptions (see Section 4.4).

```
module →                          module Communication01:
    Communication01               methods:
(int ?x, int !y) {                  method_x:
  pause(x);                           external'x: input int
  if( x > 0) {                        when: pc'thr000==1|pc'thr000==3
    y = 2 * x;                          True => x = external'x
    pause(y);                           pc'thr000==1&0<x => y.1 = x*2
  } else {                              pc'thr000==1&0<x => next(pc'thr000) = 2
    pause(x);                           !(0<x)&pc'thr000==1 => next(pc'thr000) = 3
    y = 3 * x;                          pc'thr000==1 => next(y) = y.1
  }                                     pc'thr000==3 => y.1 = x*3
}                                       pc'thr000==3 => next(y) = y.1
                                    method_y:
                                      external'y: output int
                                      when: pc'thr000==2
                                        True => external'y = y
```

Fig. 4. Example `Communication01`

The depth of the variable scope is transparent for the compilation. Finally, the *try-catch* statement is compiled by first compiling its body and the exception handler. The exit condition of the body is used to trigger a new rule, which contains the surface of the exception handler. For details, also see Section 4.4.

After compiling the surface and the depth of the program, the results are assembled in a post-processing step, before they are finally written to a file in our intermediate format. In this step, the $\mathcal{Y}$ map is used to synchronize the rules $\mathcal{R}$: for each local variable x, all rules starting at a $\texttt{pause}(x)$ are merged to a single rule so that their execution is synchronized. For each variable x which is contained in the system interface, a method (see Section 3) is created, which additionally handles the communication with the environment.

Figure 4 gives an example. For the interface variables x and y the compiler creates two methods. Each one exchanges data with the environment. The method for x is called at the first and third *pause* statements ($\mathsf{pc}_{t_0} = 1 \vee \mathsf{pc}_{t_0} = 3$), and the method for y is called at the second one.

4.2 Synchronous Actions

All synchronous guarded actions within a rule are executed simultaneously. However, the compilation algorithm presented in the previous section has to put several assignments to the same variable in a single rule. This is a problem since there can only be a single action in a rule writing a variable. Furthermore, there is another problem related to the control-flow conditions φ and T. The naive collection of conditions over the whole body of a rule does not consider intermediate updates to variables. For example, consider the I condition for the statement $\texttt{if}\ (x)\ x = \mathsf{false}; \texttt{else pause}(); \texttt{if}\ (x)\ \texttt{pause}()$ would be $x \wedge \neg x = \mathsf{false}$ in this case, which is obviously wrong. To handle both problems, several updates within a single rule must be prevented.

```
module RuleBody01
( int ?x, int !y ) {
   int a;
   pause;
   a = 101;
   if(a == x) {
      a = a + 102;
      if(a == 2*x)
         a = a + 103;
      else
         a = a + 104;
   } else
      a = a + 105;
   y = a;
}
```

```
module RuleBody01:
   init:
      True => next(a) = 0
      True => next(pc'thr000) = 1
   rules:
      pause001: when(pc'thr000==1)
         True => a.1 = 101
         x==a.1 => a.2 = a.1+102
         x==a.1&a.2==x*2 => a.3 = a.2+103
         !(a.2==x*2)&x==a.1 => a.3 = a.2+104
         !(x==a.1) => a.2 = a.1+105
         True => next(a) = x==a.1?a.3:a.2
         True => next(y) = a.4
```

Fig. 5. Example `RuleBody01`

A general solution to this problem is the *static-single assignment* (SSA) form of a program [4], where each variable is only written once. Transforming a program to SSA involves the replication of variables and the introduction of so-called *phi-functions*, which determine the current value of a variable in the case that several assignments to it have been executed before. The surface compilation algorithm generates an SSA-like form for each rule on the fly. Thereby it makes use of the substitution ρ, which stores the current mapping of program variables to incarnations. Each time an assignment is compiled in the surface, a new variable incarnation is created, and the new mapping is stored in ρ. Thereby, the different values for a variable in the course of a rule can be distinguished. Since the SSA form is generated for the body of the rule, the final update must be written back to the original variable at the end of the step so that the next rule can access it. Our compiler accomplishes that by adding a delayed assignment of the original variable. The additional intermediate variables have no impact on the performance of the final implementation: since they only need to store values within a step, hardware synthesis can map them to wires and software synthesis to local variables.

However, the body of the rule is generally not generated from its beginning to its end, e.g., if the rule starts in a branch of an *if* statement and ends outside. In this case, two parts are created separately, and the depth compilation of the sequence assembles them. It increases all incarnations of the second part to the highest one of the first part so that no collisions occur.

Figure 5 gives an example. For the program on the left-hand side, the rules shown on the right-hand side are generated. Since the program variable a has several values within the same step, the compiler generates several additional incarnations (namely $a.1$, $a.2$ and $a.3$) to distinguish them. The final value is written back to a by a delayed action at the end.

Our algorithm takes all actions between two consecutive synchronization points and combines them into a single rule. Therefore, the size of the rules may become larger than desired, and code duplication occurs since actions after control-flow joins of the program appear in several rules (the resulting code is quadratic in the worst case). This effect could be avoided by adding a `pause` statement without parameters (internal transition), which does not change the behavior of the program but causes a split up of the rules.

In principle, by adding implicit `pause` statements at the control-flow joins, any code duplication could be avoided. In an extreme version, these additional positions can be introduced after each assignment so that the whole SSA computation would not be needed anymore. In the compiler, this could be accomplished by returning the instantaneity condition false at the desired points. Then, a new position is created, where the control flow can resume in the next step. We finally decided against the introduction of these implicit `pause` statements since they may make the program more sequential than desired, and only an additional and possibly expensive analysis [11] can reconstruct the lost parallelism. However, the programmer can use them in the source code to explicitly split up steps.

4.3 Shared Variables and Synchronization

As already pointed out in Section 2.2, only a single thread has write access to the current value of a variable x. In order to avoid race conditions, all other threads only know the value that the variable had at its previous synchronization point. Hence, our translation also needs several variables in the generated code to store the different values that a program variable x has in the individual threads.

Obviously, the first question is how many variables are needed for a program variable x. For a set of parallel threads on the same level, two variables suffice: the first one is attributed to the writer, and it stores the current value of x. In addition, we need a copy, which stores the value known at the last synchronization point and which is the same for all threads on the same level. In the general case, we need n copies, where n is the maximum nesting of parallel statements.

Thus, we always create an additional copy when a new parallel statement starts. Note that we do not need to create copies for all variables but only for variables which are written by some thread and read by another one. In the translation shown above, the surface part of the *parallel* initializes the copies, while the depth part forwards the mapping to the subthreads via their parameter ρ.

The copies are updated at each synchronization point. This is accomplished by a post-processing step in our translation algorithm, which merges all rules starting at synchronization points for a given variable x, and generates the appropriate copy actions.

The right-hand side of Figure 1 shows the rules for the given program, which has already been explained in Section 2.2. For the shared variable a, a

```
module Exception03
(bool ?x) {
  int x1; int x2;
  int i;
  try {
    {
      x1 = i;
      throw T;
    } || {
      try{
        {
          x2 = i;
          throw U;
        } || {
          pause(i);
        }
      } catch(U)
        {nothing;}
      i = i + 1;
      pause(i);
      i = i + 1;
    }
  } catch (T)
    {nothing;}
}
```

```
module Exception03:
  init:
    ec'thr000==0 | ec'thr000>1 => next(ec'thr000) = 1
    ec'thr000==0 | ec'thr000>1 => next(pc'thr001) = −1
    ec'thr002==0 | ec'thr002>2 => next(ec'thr002) = 2
    ec'thr002==0 | ec'thr002>2 => next(pc'thr003) = −1
     ...
  rules:
    pause001: when(pc'thr004==1)
      True => next(i) = i+1
      True => next(pc'thr002) = 2
    prop003: when(
        pc'thr003==−1 & pc'thr004==1)
      True => next(pc'thr004) = −1
    exit004: when(
        pc'thr003==−1 & ec'thr003==0 &
        & pc'thr004==−1 & ec'thr004==0 )
      True => next(pc'thr002) = −1
    catch005: when(pc'thr002==−1&2==ec'thr002)
      True => next(i) = i+1
      True => next(pc'thr002) = 2
    pause006: when(pc'thr002==2)
      True => next(i) = i+1
    prop008: when(pc'thr001==−1 & pc'thr002==2)
      True => next(pc'thr002) = −1
    catch010: when(pc'thr000==−1 & ec'thr000==1)
```

Fig. 6. Example `Exception03`

copy $a@1$ is created in the pre-processing step, which is used by the reading threads (as the second last action of rule a shows). The compilation of surface and depth generates a separate rule for each `pause`(a) statement. In the post-processing step, these rules are merged to a single rule a. In this rule, the current value of a is transmitted (first action of rule a) to the copy $a@1$ so that all other threads have access to the new value.

4.4 Best-Effort Exceptions

Finally, the last problem that the compiler has to tackle is the compilation of exceptions. Basically, this task is to generate code for three situations: (1) exceptions are thrown by some thread (2) they are propagated to other threads running in parallel and to their parent thread, and (3) they must be finally caught.

Part (1) is handled by the compilation of the `throw` e statement. In the surface compilation, the program counter of the current thread is set to the exit position -1. The code ec_t is examined, where t is the thread in which the exception e is declared (not necessarily the same thread as the emitter). It is overwritten (and set to e) if e is of higher priority than the currently scheduled exception

(where exceptions with greater scope have higher priority). Thereby, the exception gets scheduled, and its handler will be invoked when/if its body is finally aborted.

Part (2) is handled by the depth compilation of the parallel statement. There, actions are generated, which cause that concurrent subthreads exit if they communicate with a thread that has already exited. They basically state that whenever a thread has exited and another one, which terminates normally or tries to synchronize with it over a common variable, the latter one will also exit. Additionally, an exit rule is created that causes an exit of the current thread if all subthreads have exited.

Part (3) is finally handled by the compilation of `try` S_1 `catch`(e) S_2. The exception handler S_2 will be invoked if the body S_1 has exited and if the exit code ec_t of the current thread t is the given exception e.

Figure 6 gives an example for the compilation of the exceptions. The different kinds of rules that are generated can be distinguished by their names. The *prop* rules are responsible for the propagation of exceptions between concurrent threads, while *exit* rules forward the exit of subthreads to its parent. As expected, the *catch* rules implement the trigger of the exception handler. In the example, *prop003*, *exit004* and *catch005* implement exception U, while *prop008* and *catch010* are responsible for exception T. For example, rule *prop003* handles the case where the first thread of the inner parallel statement has exited ($\mathsf{pc}_{t_3} = -1$) and the second one is at the pause statement ($\mathsf{pc}_{t_4} = 1$, where 1 is the label automatically assigned to the first synchronization point in the thread): then, the second thread also exits ($\mathsf{pc}_{t_4} = -1$). Rule *exit004* checks whether these both subthreads have exited and no exception can be caught in their contexts ($\mathsf{ec}_{t_3} = 0$ and $\mathsf{ec}_{t_4} = 0$). In that case, the parent thread also exits. Rule *catch005* finally catches the exception. If the corresponding thread has exited and its exit code matches the one of the exception (2 represents exception U in our case), the exception handler is executed.

5 Experimental Results

The compilation algorithm presented in the previous section generated asynchronous guarded actions for a given Emerald program. In the following, we show how this result can be used by other backends. Thereby, we focus on two targets, which have not been considered for SHIM before, namely hardware synthesis and model checking of program properties by SMV.

For both targets, we first translate the asynchronous guarded actions to synchronous ones as described in [2]. This step includes the generation of a scheduler, which fires a subset of the activated guarded actions. Since our asynchronous guarded actions are scheduling independent (as already exploited in [17,18]), we can skip the conflict analysis described in [2] and add a simple ASAP scheduler, which immediately fires all activated actions. However, in order to optimize the system according to a given metric, another scheduler could be added, which is the core of the BlueSpec approach[3] [11].

[3] `http://www.bluespec.com/`

Example	Emerald		IR	Verilog FPGA Implementation				
	threads	.emd	.aif	.v	FFs	LUTs	DSP48s	BUFGs
ProdCons	3	20 loc	19 kB	8 kB	13	24	0	1
Lattice	7	59	57	17	28	187	4	1
TokenRing	17	110	108	35	28	106	0	1
Eratosthenes	23	338	403	113	138	660	20	0
Green	11	199	611	168	1358	2949	0	1

Fig. 7. Experimental Results: Hardware Synthesis

For both targets we used the following set of examples (see Figure 7): *ProdCons* is a simple producer-consumer system, where the producer terminates after 256 elements so that the consumer is blocked. *Lattice* is a naive parallel implementation of a lattice filter. *TokenRing* is a model of token ring consisting of 16 stations. *Eratosthenes* is a pipelined parallel prime searching algorithm. Finally, *Green* is a straightforward implementation of Green's parallel sorting network.

For the hardware synthesis, we used the Verilog backend of our Averest toolset to generate structural HDL code. We use the balanced synthesis strategy of Xilinx's ISE Design Suite to synthesize the examples for a Xilinx Virtex-4 XC4VLX15 board.

The table in Figure 7 gives the name and the size of the orignal Emerald source file and the size of the XML-based intermediate code generated by our translation. The middle part shows the size of the generated Verilog code, and the resources needed to implement the design on the FPGA board (flip-flops, look-up tables, arithmetic units and global buffers). Apparently, all examples can be implemented with a modest amount of resources. It can be also noticed that the AIF and Verilog files are bigger than the original source and the final design. Since the current version of our compiler structurally translates the source programs without much optimization, it may contain dead code. For example, exception propagation rules are generated even if no exception can be thrown. Nevertheless, the constant propagation and logic pruning steps of the hardware synthesis detect and eliminate these parts so that no hardware is generated for them.

As a second backend, we connected SMV for a formal verification of program properties by symbolic model checking. Again, we use the existing infrastructure of our Averest toolset to transform the synchronous guarded actions into a transition system read by SMV. From the technical side, we can translate the complete data-flow of the program (since we used Averest expressions in Emerald programs) and handle all SMV specifications. However, as we want to check properties of the original SHIM program, only scheduling independent properties make sense, e.g., we canot combine variables of concurrent threads in a propositional formula or use X in specifications.

Example	Emerald		IR	SMV			
	threads	.emd	.aif	.smv	states	nodes	time
ProdCons	3	20 loc	19 kB	50 kB	179	14,849	0.72 s
Lattice	7	59	57	148	136	129,242	8.13
TokenRing	17	110	108	111	587	14,910	48.26
Eratosthenes	23	338	403	346	1329	5,544,291	115.28
Green	11	199	611	443	534	151,008	1915.73

Fig. 8. Experimental Results: Model Checking

In contrast to previous work [17], we do not create designated model for a specific analysis but a general one that can be used to verify all properties SMV can handle. We verified for the given programs the properties described in the following. As already presented in [17], deadlock detection is an interesting static analysis for CSP-based programs. We can mimic this analysis by generating an assertion that checks whether there will be at least one activated rule as long as the program is not terminated. Let γ_i be the guards of all asynchronous guarded actions and ℓ_T the termination position of the main thread t_0 determined by our compilation algorithm, then we check

$$\mathsf{AG}\left((\mathsf{pc}_{t_0} = \ell_T) \vee \bigvee_i \gamma_i\right)$$

Another analysis that is very useful in the context of Emerald is to check whether a program throws exceptions. In that case, another interesting property is whether thrown exceptions are eventually caught (thus, whether best-effort preemption is successful). In our model, an exception is thrown by writing its error code to ec_t. Eventually, thread t must either catch this exception or another exception of higher priority (its exit code is greater) is thrown. For an exception e declared in t with error code i and a handler at program position p, this can be encoded as follows ($\underline{\mathsf{U}}$ is the strong until operator):

$$\mathsf{A}\left[\mathsf{ec}_t = i \mathbin{\underline{\mathsf{U}}} \mathsf{ec}_t > i \vee \mathsf{pc}_t = p\right]$$

The table in Figure 8 summarize the results for verifying these properties on a recent MacMini. In order to verify all properties, we abstracted the data-flow of the given program as much as possible in order to restrict the state space to a manageable size. The size of the SMV input files, as well as its memory consumption and run-time is given. It can be seen that the properties can be verified for the given examples with reasonable effort.

6 Conclusions

In this paper, we presented a translation of Emerald programs to asynchronous guarded actions. The translation reduces the complex semantics on the source

code level to a relatively simple model on the intermediate level, which still allows efficient analysis and synthesis. By using asynchronous guarded actions as a target, many existing techniques and tools can be applied to SHIM and Emerald programs. In particular, hardware synthesis and verification based on temporal logics become possible. An interesting project for the future might be the port of the old design flow to asynchronous guarded actions so that the other existing compilation approaches can be validated and tighter coupled to the new one.

References

1. Brandt, J., Schneider, K.: Separate compilation for synchronous programs. In: Falk, H. (ed.) Software and Compilers for Embedded Systems (SCOPES). ACM International Conference Proceeding Series, vol. 320, pp. 1–10. ACM, Nice, France (2009)
2. Brandt, J., Schneider, K., Shukla, S.: Translating concurrent action oriented specifications to synchronous guarded actions. In: Lee, J., Childers, B. (eds.) Languages, Compilers, and Tools for Embedded Systems (LCTES). pp. 47–56. ACM, Stockholm, Sweden (2010)
3. Chandy, K., Misra, J.: Parallel Program Design. Addison-Wesley, Austin, Texas, USA (May 1989)
4. Cytron, R., Ferrante, J., Rosen, B., Wegman, M., Zadeck, F.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems (TOPLAS) 13(4), 451–490 (October 1991)
5. Dijkstra, E.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM (CACM) 18(8), 453–457 (1975)
6. Dill, D.: The Murphi verification system. In: Alur, R., Henzinger, T. (eds.) Computer Aided Verification (CAV). LNCS, vol. 1102, pp. 390–393. Springer, New Brunswick, New Jersey, USA (1996)
7. Edwards, S., Tardieu, O.: SHIM: a deterministic model for heterogeneous embedded systems. In: Wolf, W. (ed.) Embedded Software (EMSOFT). pp. 264–272. ACM, Jersey City, New Jersey, USA (2005)
8. Edwards, S., Tardieu, O.: Efficient code generation from SHIM models. In: Irwin, M., De Bosschere, K. (eds.) Languages, Compilers, and Tools for Embedded Systems (LCTES). pp. 125–134. ACM, Ottawa, Ontario, Canada (2006)
9. Edwards, S., Vasudevan, N., Tardieu, O.: Programming shared memory multiprocessors with deterministic Message-Passing concurrency: Compiling SHIM to Pthreads. In: Design, Automation and Test in Europe (DATE). pp. 1498–1503. IEEE Computer Society, Munich, Germany (2008)
10. Hoare, C.: Communicating Sequential Processes. Prentice Hall (1985)
11. Hoe, J., Arvind: Operation-centric hardware description and synthesis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD) 23(9), 1277–1288 (September 2004)
12. Järvinen, H., Kurki-Suonio, R.: The DisCo language and temporal logic of actions. Technical Report 11, Tampere University of Technology, Software Systems Laboratory (1990)
13. Schneider, K.: The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany (December 2009)

14. Tardieu, O., Edwards, S.: R-SHIM: deterministic concurrency with recursion and shared variables. In: Hoe, J., Palsberg, J. (eds.) Formal Methods and Models for Codesign (MEMOCODE). pp. 202–202. IEEE Computer Society, Napa, California, USA (2006)
15. Tardieu, O., Edwards, S.: Scheduling-independent threads and exceptions in SHIM. In: Min, S., Yi, W. (eds.) Embedded Software (EMSOFT). pp. 142–151. ACM, Seoul, South Korea (2006)
16. van Berkel, K.: Handshake Circuits. Cambridge University Press, Cambridge, Great Britain (1993)
17. Vasudevan, N., Edwards, S.: Static deadlock detection for the SHIM concurrent language. In: Formal Methods and Models for Codesign (MEMOCODE). pp. 49–58. IEEE Computer Society, Anaheim, California, USA (2008)
18. Vasudevan, N., Edwards, S.: Buffer sharing in CSP-like programs. In: Bloem, R., Schaumont, P. (eds.) Formal Methods and Models for Codesign (MEMOCODE). pp. 151–160. IEEE Computer Society, Cambridge, Massachusetts, USA (2009)