Modular Code Generation for SCCharts

Gavin Lüdemann

Bachelor's Thesis
September 2021

Real-Time and Embedded Systems Group
Prof. Dr. Reinhard von Hanxleden
Department of Computer Science
Kiel University

Advised by
M.Sc. Alexander Schulz-Rosengarten
and Dr.-Ing. Steven Smyth

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

Modularization is a central concept of software development. It enables programmers to use well-documented and tested code instead of solving every problem from scratch. The synchronous language SCCharts has a modularization system called *Referenced SCCharts* that allows a model to be included by another model. Its current implementation does not, however, support *modular code generation*: a model cannot be compiled separately from the modules that it uses.

This thesis aims to enrich SCCharts by introducing Module Call Semantics (MCS), an implementation of Referenced SCCharts amenable to modular code generation, to the language. MCS allows SCCharts to be compiled into self-sufficient units that can be called by any other model. Module behavior is executed atomically and scheduled using a black-box approach. In this thesis, the concepts behind MCS are presented and its implementation into KIELER is documented. The run-time performance and model size implications of MCS are benchmarked and analyzed.

Acknowledgements

I want to thank Alexander Schulz-Rosengarten and Dr.-Ing. Steven Smyth, in no particular order, for advising this thesis, providing valuable feedback on both my methods and my writing, and gently reminding me of the finitude of time whenever necessary. Furthermore, I want to thank Prof. Dr. Reinhard von Hanxleden for giving me the opportunity to write this thesis, as well as his detailed feedback and valuable advice.

Finally, I want to thank all attendees of the Real-Time and Embedded Systems Group's daily virtual morning tea for helping to keep me sane while working on this thesis during these times of social distancing.

Contents

List of Figures

List of Tables

List of Abbreviations

MES	Module Expansion Semantics
MCS	Module Call Semantics
MoC	Model of Computation
SCMoC	Sequentially Constructive Model of Computation
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
SCCharts	Sequentially Constructive Statecharts
iur	initialize-update-read
SCG	Sequentially Constructive Graph
SC	Sequentially Constructive
LoC	Lines of Code
CLI	Command Line Interface
KiCo	KIELER Compiler
IQR	Interquartile Range
CFS	Complex Final State
RCP	Rich Client Platform

Chapter 1

Introduction

Embedded systems are ubiquitous. Every car, train and airplane produced today relies on them for control and communication. Even small household appliances commonly use embedded systems. A key challenge of such systems is the handling of concurrency.

It is arguably often easiest to model time-independent tasks as concurrent. A popular concurrency model is the *thread* model: each task has a set of instructions that operate independently on shared variables. These shared variables are used to communicate results between threads. However, there is no inherent guarantee that the order of execution of these threads does not influence their results. In order to avoid such *race conditions*, traditional imperative languages like C and Java employ synchronization techniques such as *barriers* and *mutual exclusion* that partially constrain the order of operations. However, the correct usage of such techniques is generally left to programmers.

In order to address these issues in reactive systems, synchronous languages like Esterel [Ber00] and LUSTRE [CPH+87] and, more recently, Blech [GG18] and SCCharts [HDM+13] were introduced. They divide program execution into *reactions* (or *ticks*). In a tick, inputs are read and a new internal state, as well as outputs, are computed. A sequence of ticks is called a *trace*. By definition, synchronous languages encode *determinate* programs. A program is determinate if for every program trace, i. e., equal inputs produce equal outputs. In Esterel and many other synchronous languages, a variable is restricted to have exactly one value in each tick.

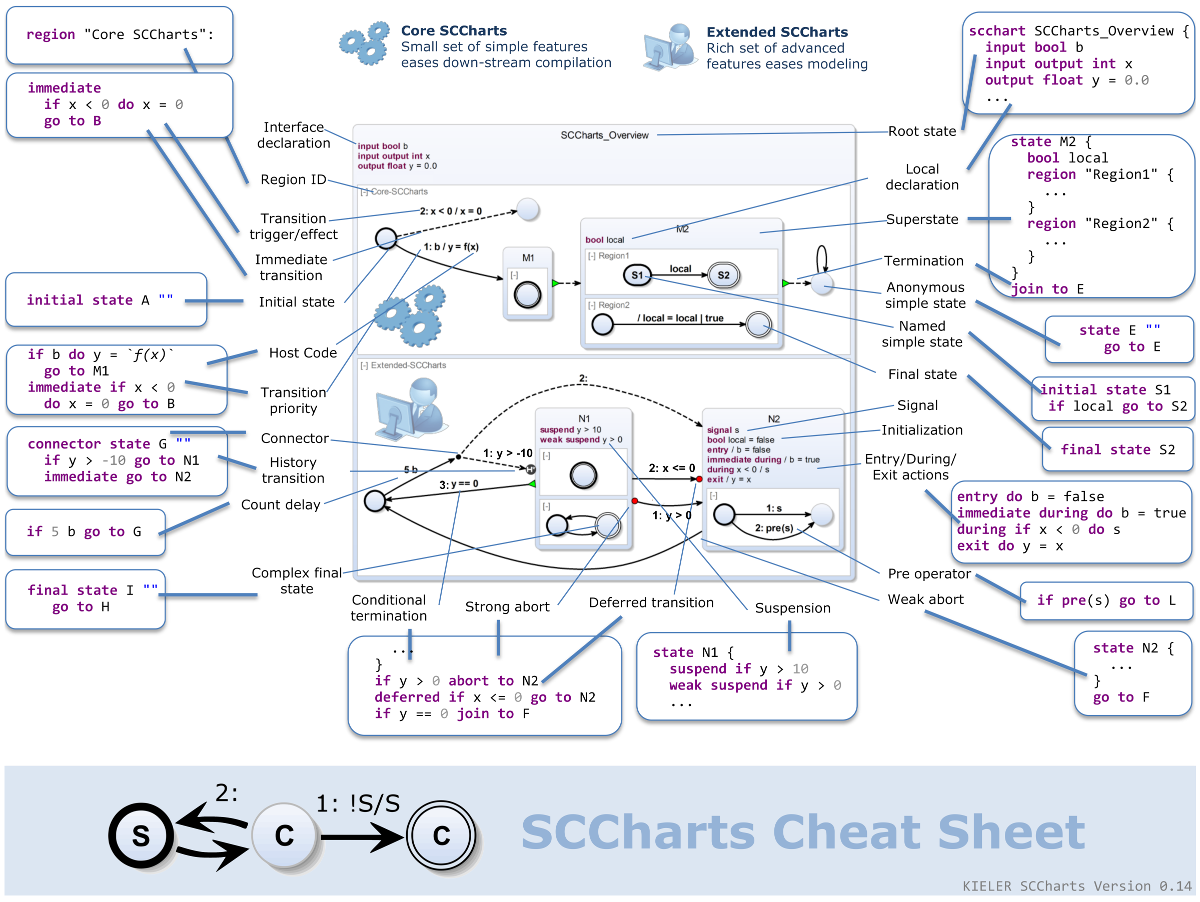
1.1 Sequentially Constructive Statecharts

The Sequentially Constructive Model of Computation (SCMoC) [HMA+14], is a conservative extension of the synchronous Model of Computation (MoC), i. e., all *constructive* programs as defined by the synchronous MoC are also Sequentially Constructive (SC) and retain their semantics under the SCMoC. Additionally, variables are allowed to have multiple values per tick, provided a schedule that ensures determinacy can be found. In particular, sequentially ordered accesses on the same variables are always permitted.

Sequentially Constructive Statecharts (SCCharts) [HDM+13] is a graphical synchronous language semantically based on the SCMoC. Its visual syntax is closely related to *SyncCharts* [And03] and *StateCharts* [Har87]. SCCharts can be modeled using a dedicated textual syntax. The language's more permissive scheduling thanks to sequential constructiveness allows the modeler to use more traditional, imperative constructs prohibited in other synchronous languages. This makes SCCharts more accessible to programmers experienced with developing software in languages like C or Java. An overview of SCCharts'features is shown in Figure 1.1.

1.1.1 Module Expansion Semantics

Like many other programming languages, SCCharts provides a mechanism for modularization. It is known as *referenced SCCharts*. Referenced SCCharts were first described in the *Railway Project Technical*



```
region "Core SCCharts":
```

```
immediate
  if x < 0 do x = 0
  go to B
```

```
initial state A ""
```

```
if b do y = `f(x)`
  go to M1
immediate if x < 0
  do x = 0 go to B
```

```
connector state G ""
  if y > -10 go to N1
  immediate go to N2
```

```
if 5 b go to G
```

```
final state I ""
  go to H
```

```
  ...
}
if y > 0 abort to N2
deferred if x <= 0 go to N2
if y == 0 join to F
```

```
state N1 {
  suspend if y > 10
  weak suspend if y > 0
  ...
```

```
scchart SCCharts_Overview {
  input bool b
  input output int x
  output float y = 0.0
  ...
```

```
state M2 {
  bool local
  region "Region1" {
    ...
  }
  region "Region2" {
    ...
  }
}
join to E
```

```
state E ""
  go to E
```

```
initial state S1
  if local go to S2
```

```
final state S2
```

```
entry do b = false
immediate during do b = true
during if x < 0 do s
exit do y = x
```

```
if pre(s) go to L
```

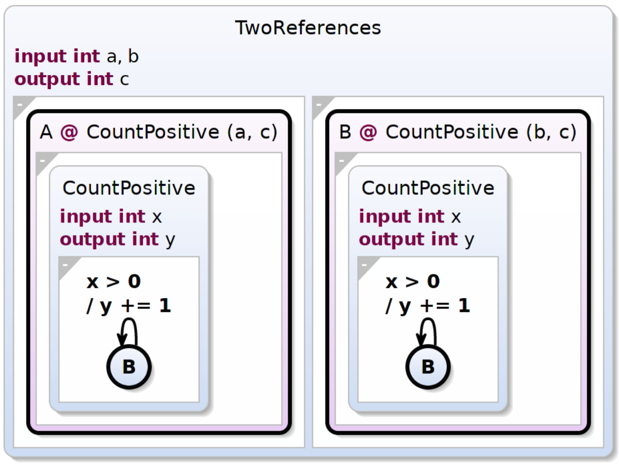
```
state N2 {
  ...
}
go to F
```

Figure 1.1. The SCCharts Cheat Sheet gives an overview of the language's features as well as textual and visual syntax [SMS+19].

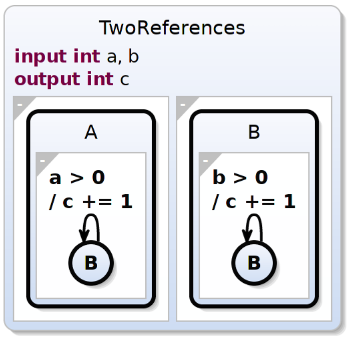
Report [SMS+15] as one of the SCCharts extensions necessary in order to develop an SCCharts model as large as the *Railway Controller* the report centers around. In addition to the root model, an arbitrary number of additional models can be defined, either in the same or in additional source files. A model can then have references to these models, which contain the referenced model's name and a variable binding specification as seen in Figure 1.2a. Any valid SCCharts model can be referenced this way.

The existing implementation of referenced SCCharts is provided by Module Expansion Semantics (MES). MES is closely related to C-style macro expansion in that it is purely a replacement system: all references in the root model are replaced by the content of the referenced model, the local variables of which are then renamed to conform to the binding specification. Figure 1.2b shows how the variables are renamed. If the referenced models themselves contain references, they are recursively resolved.

Because references are expanded into the root model before an execution schedule is determined, it is possible for the compiler to arbitrarily interleave concurrent statements from all models. This *white-box scheduling* allows the compiler to find schedules for models that it could not find if referenced models were a black box.



(a) Example for referenced SCCharts: the model has two references, A and B, to the same model, CountPositive. The first reference binds a to x, the second binds b to x. Both bind c to y. In effect, the model counts the cumulative number of positive integers in two input streams.



(b) The same model after reference expansion

Figure 1.2. An example for referenced SCCharts and Module Expansion Semantics (MES)

1.2 Problem Statement

The strong coupling between root and referenced models using MES has a number of disadvantages. Every reference to another model increases the root model's size by the size of the referenced model. Hence, there is no economy of scale regarding multiple instances of the same behavior. Because of white-box scheduling, referenced models cannot be compiled in separation from the root model. White-box scheduling also leads to the possibility that purely structural changes to a referenced model that do not modify external behavior drastically alter the whole model's scheduling requirements—or even make it no longer SC and therefore unschedulable. This behavior is problematic w.r.t. encapsulation. Hence, in order to support modular code generation, SCCharts needs a modularization system that does not rely on white-box scheduling.

1.3 Contributions

This thesis proposes Module Call Semantics (MCS) as an alternative reference resolution approach that enables modular code generation to complement MES. An implementation into Kiel Integrated

Environment for Layout Eclipse Rich Client (KIELER) supporting the *netlist-based C* compiler is given and its size and execution time performance versus MES is evaluated. Advantages and limitations of MCS are discussed and recommended use cases are derived.

1.4 Related Work

In synchronous languages, there are multiple different approaches to modularization. These can be categorized by the compiler's knowledge of the inner workings of modules:

▷ White-box approaches like Esterel modules [Est08] and SCCharts MES [SMS+15; HDM+13] have full knowledge of every module. In Esterel and SCCharts MES, modules are expanded into the main model, similar to macro expansion in C [SW87]. Therefore, the entire source code of every module must be available at compile time.

  White-box scheduling is the most permissive approach. However, it is not applicable to modular code generation because it requires all modules to be compiled together with the main model.

▷ Gray-box approaches, for which the compiler is provided with *some*—usually automatically generated—scheduling information, have been described by Hainque et al. [HPL+99] for Esterel and Lublinerman et al. [LST09] for SCADE and Simulink.

  There are many different gray-boxing techniques. They generally trade computational complexity during the compilation of modules for more permissive scheduling. Arguably, MCS applies very primitive gray-boxing by providing some high-level scheduling information, such as whether or not a module can terminate instantaneously.

▷ Gretz et al. [GGM+20] formalized the semantics for black-box scheduling. This allows the inclusion of modules without any knowledge of internal behavior. Recently, Smyth [Smy21] proposed a black-box approach for compiling SCCharts. MCS is based on that approach.

1.5 Outline

The remainder of this thesis is organized as follows. Chapter 2 introduces MCS and its underlying concepts, along with the necessary preliminaries on SCCharts. In Chapter 3, the implementation details associated with MCS are documented. The newly introduced model processors are shown and put into context with KIELER as a whole. Chapter 4 evaluates both the concept of MCS and the approach taken in this thesis, demonstrating their conceptual and implementation based limitations and advantages. As a central part of this chapter, performance measurements on several models, both synthetic and real-world are given and analyzed. In closing, Chapter 5 gives a summary of the results of the thesis, as well as pointing out future work.

Chapter 2

Modular Code Generation

The ability to generate code from SCCharts in a modular fashion is desirable for multiple reasons. It enables more efficient build processes, easier unit testing and the distribution of SCCharts-based libraries without disclosure of the source code. This chapter presents the Module Call Semantics (MCS), which allows modular code generation in SCCharts. Its concepts are laid out in Section 2.2 and its integration into SCCharts is explained in Section 2.3. Section 2.4 explains how separate compilation is supported. In Section 2.5, MCS is compared to *call-by-value* argument handling. Finally, Section 2.6 discusses the possible implications of MCS for run-time performance.

2.1 Preliminaries

The concepts and design decisions behind MCS depend on details about the features and compilation of SCCharts described in this section. Furthermore, the following sections use terminology specific to SCCharts and MCS. This terminology is defined here.

2.1.1 SCCharts Preliminaries

SCCharts is characterized by a hierarchical set of features: every model's behavior can be expressed in terms of a small set of model elements called *Core SCCharts*. These are the basis for *Extended SCCharts*, which encompass all of the language's high-level features. All extended features are defined in terms of their transformation into either extended or core features [Mot17].

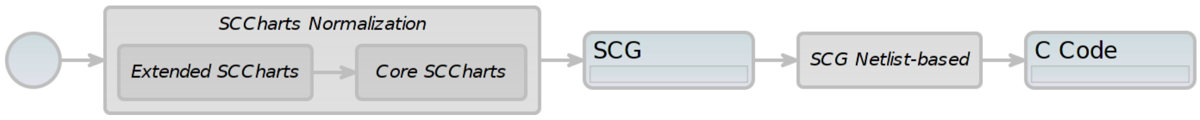


Figure 2.1. The *netlist-based C* compiler pipeline

The compilation of SCCharts models is achieved in multiple stages. Figure 2.1 gives an overview of the netlist-based C compiler pipeline, which is discussed here. In the first stage, extended features are transformed in a series of model-to-model transformations. Each extended feature is associated with a corresponding transformation processor. A model that has been processed by such a processor does not use its associated feature any more, neither does it use features eliminated by previous transformations. In this manner, a model is passed though each processor exactly once. The result is a model that uses only Core SCCharts features.

The second stage is normalization. During normalization, model elements are transformed such that only a five distinct patterns remain: *Thread*, *Parallel*, *Conditional*, *Assignment* and *Delay*. Each of these can then easily be mapped to exactly one element of a Sequentially Constructive Graph (SCG). The SCG is a graph consisting of statement nodes and control-flow edges. It lends itself to determining data dependencies and finding an execution schedule. Note that not every code synthesis uses the SCG as an intermediary stage.

The execution schedule is determined as follows. First, sequential statements are scheduled in sequence. Second, concurrent writes are scheduled according to the initialize-update-read (iur) protocol. The iur protocol specifies that absolute write operations (e. g., `x = true`) are scheduled before relative writes using infix notation (e. g., `x |= true`), which in turn are scheduled before read operations (e. g., `if x`). Third, if multiple concurrent absolute (resp. relative) writes occur, they must be *confluent*, i. e., the result does not depend on the order of execution. If they are not, the model is rejected.

The final stage of compilation is code generation. Three functions are generated: `tick`, `reset` and `logic`. The tick function is called by the host software in order to compute the model's reaction for a single tick. The reset function is called by the host software before the first tick. It initializes the model. The logic function implements most of the model's behavior. However, because it is generally only called from the tick function, it is conflated with the tick function for the remainder of this thesis.

By design, the state of *all* internal variables (including *guards*, registers used to determine control flow) is computed in every tick of a program compiled using the netlist-based synthesis. This generally leads to more predictable execution times for each tick, e. g., when compared to the priority-based approach [Pei17].

2.1.2 Additional Terminology

In the following, a *module* is an SCCharts model that provides some functionality to another model. The two models are in a *caller-callee* relation. A caller can rely on an arbitrary number of callee modules. Every callee can also be a caller to another module. If a caller holds multiple references to the same module, each of these references is an *instance* of the module.

2.2 Module Call Semantics

The alternative to MES presented in this section is to not expand a referenced SCChart into the root model. Instead, its behavior is compiled into a separate source file containing a tick function and a reset function that can then be called from within any other model. Additionally, each instance of a module has its own datastructure to hold its variables.

MCS is not meant replace MES. Rather, it is meant to be an alternative explicitly chosen by the modeler. They should be able to make this choice for each individual reference. Therefore, it makes sense to make the distinction between the two a grammatical feature, not a compiler option. Furthermore, substituting one approach for the other for any given reference should not require large amounts of text editing. Consequently, it was decided to syntactically distinguish the two approaches by using the new **`calls`** keyword in place of the **`is`** keyword in reference statements to specify a MCS reference. In the visual syntax, the **`@`** token is then also replaced by **`calls`** in order to indicate an MCS reference.

In order to make their use as intuitive as possible, it was decided that MCS references must be resolved such that the following conditions hold:

(i) The callee's tick function is called if and only if its referencing state is active within the caller model.

(ii) The callee's reset function is called if and only if its referencing state is (re-)entered, and before the tick function is called.

(iii) All concurrent (absolute and relative) writes to the callee's input variables are completed before its tick function is called.

(iv) Output values read from the callee can only be read after the callee's tick function has returned and cannot be concurrently overwritten.

(iii) and (iv) is accomplished by using assignments in order to copy values into and out of the callee's data structure. If inputs and outputs are written directly before and after each call to the tick function, the IUR protocol enforces these conditions: let a and b be variables of the caller, x an input variable and y an output variable of the callee such that a is passed to x and b is retrieved from y. Then according to the IUR protocol all concurrent absolute or relative writes to a must occur before the value of a is assigned to x. Similarly, if b is read concurrently, the IUR protocol demands that the value of y is written before it can be read. Because the assignment of y to b is an absolute write operation, all other concurrent absolute writes are prohibited. Because the copying of inputs, call to the tick function and copying of outputs are sequentially ordered, their execution order is fixed.

In order to ensure (i), the caller's reference to the callee is replaced by a complex state, the *proxy state*. The sequence of statements specified above can be placed within such that the tick function is called exactly once in each tick during which the proxy state is active. Then the proxy state must be entered in order for the tick function to be called. In SCCharts, there are two ways a complex state can be exited: by (strong or weak) abort and by termination. If the proxy state is strongly aborted, then the tick function cannot be called. Therefore, aborting the proxy state aborts the callee as required. However, termination by the callee must be handled explicitly. To that end, the callee's termination status is checked after execution of its tick function. If the callee terminates, the proxy state must also transition into a final state. Finally, (ii) can be accomplished by ensuring that the proxy state's initial state has exactly one outgoing transition that triggers a call to the callee's reset function.

2.3 Transformation Rules

This section shows how the design considerations laid out in the previous section are implemented. All existing Extended SCCharts features can be defined by their respective transformations into other SCCharts features. This is not the case for MCS, however, because passing of input and output variables, as well as calling a model's tick function, require a level of reflection that SCCharts does not provide.

It is therefore necessary to implement MCS using target language features. Nonetheless, it is best practice to operate as generically as possible. Consequently, the MCS transformation is split into two parts: the *SCCharts transformation*, which operates on an SCCharts model and is target language independent, and the *SCG transformation*, which operates on an SCG. These two caller-side transformations are sufficient in order to call any SCCharts model. The following subsections explain each transformation in detail.

2.3.1 The Module Call SCCharts Transformation

This transformation modifies an input model that uses MCS references into one that does not. However, unlike other Extended SCCharts transformations, the resulting model is not in and of itself equivalent to the input model. This is only achieved later in the compilation by the SCG transformation.

Figure 2.2 shows an SCChart that uses a single MCS reference. The caller model has two boolean variables a and b that are bound to the input x and the output y of the callee, respectively. The inner behavior of the callee does not impact the transformation in any way.

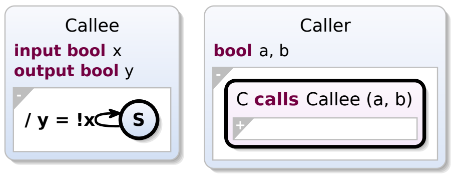


Figure 2.2. An example SCChart before transformation: the callee on the left is referenced by the caller on the right.

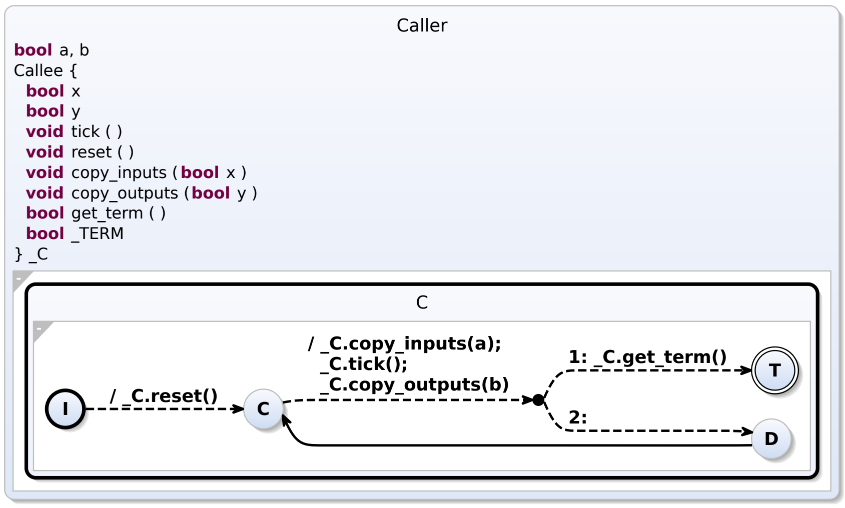


Figure 2.3. The same model as in Figure 2.2 after SCCharts transformation

The results of the transformation can be seen in Figure 2.3. First, a *proxy class* for the callee is added to the caller. The class has the following members:

▷ The callee's input and output variables;

▷ The callee's termination flag;

▷ A method declaration for the tick and reset functions; and

▷ Placeholder declarations for methods that copy inputs and outputs, as well as determine termination.

The callee's input and output variables and its termination flag, as well as the tick and reset functions, are declared in order to reference them in the proxy state. The copy_inputs, copy_outputs and get_term functions do not exist; calls to them are replaced during the SCG transformation. The signature of copy_inputs and copy_outputs contains the inputs and outputs of the callee, respectively. Note that while a proxy state exists for each instance, the proxy class is created only once for each unique callee.

Second, the reference is converted into the proxy state. The proxy state is a simple state machine that implements the considerations discussed in Section 2.2. When it is entered, the reset function is called, inputs are copied, the tick function is called and outputs are retrieved. If the callee terminates, so does the proxy state. Otherwise, control rests such that in the next tick, the tick function is called again.

Support for Non-instantaneous Modules

Because the proxy state can potentially terminate instantaneously, scheduling restrictions apply. For instance, if the state joins to itself, the compiler detects an instantaneous loop and the model is rejected. This is expected behavior if the callee model can indeed terminate instantaneously. In order to be able to have a non-instantaneous callee join to itself regardless, the `@isDelayed` annotation is introduced. If the callee is annotated with `@isDelayed`, the alternative proxy state seen in Figure 2.4 is used. As a consequence, the proxy state cannot terminate within its first tick.

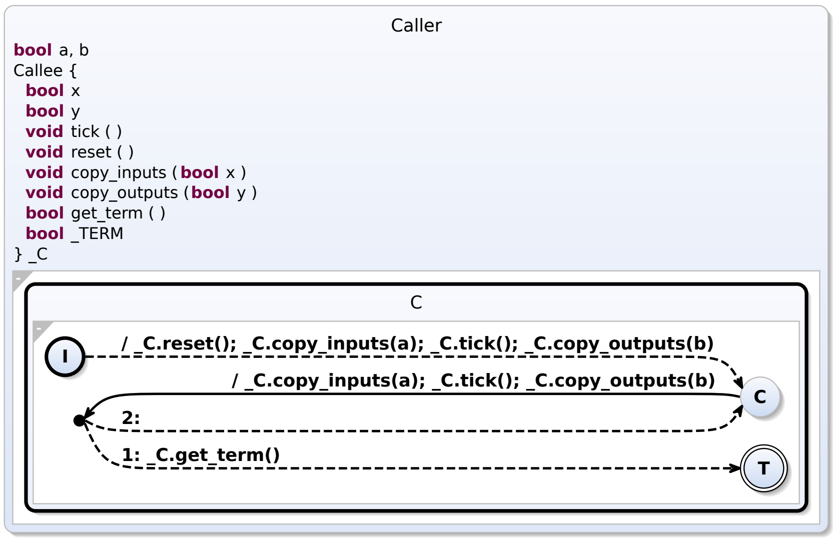

Caller
bool a, b
Callee {
bool x
bool y
void tick ( )
void reset ( )
void copy_inputs (bool x)
void copy_outputs (bool y)
bool get_term ( )
bool _TERM
} _C
C
I
/ _C.reset(); _C.copy_inputs(a); _C.tick(); _C.copy_outputs(b)
/ _C.copy_inputs(a); _C.tick(); _C.copy_outputs(b)
C
2:
1: _C.get_term()
T

Figure 2.4. The model from Figure 2.2 after the alternative, non-instantaneous transformation

Support for Complex Final States.

A Complex Final State (CFS) is a final state with internal behavior or outgoing transitions. CFSs present a challenge to MCS because the proxy state normally cannot transition out of its final state. Therefore, if the callee finishes any tick in a final state, it remains there until re-entered. This conserves computational resources by not executing a callee that has already terminated. However, if a callee is no longer executed after reaching a CFS, it has no effect. In order to allow CFS to be used in callee models, the `@nonFinal` annotation is introduced. It adds an outgoing transition to the proxy state's final state, allowing execution to be resumed after callee termination. The resulting proxy state is shown in Figure 2.5. Note that `@nonFinal` can be combined with `@isDelayed`. Figure 2.6 shows the resulting proxy state.

2.3.2 The Module Call SCG Transformation

This transformation takes the output of the existing SCCharts to SCG transformation as its input. Figures 2.7a and 2.7b show the SCGs generated from the example model by this SCCharts to SCG transformation. Note that during simultaneous compilation (as detailed in the next section), one SCG is generated for each model. Both generated SCGs are shown here: the callee SCG and the caller SCG.

As mentioned above, the module call SCG transformation is target language dependent. Note that while generation of Java code is currently unsupported, the transformation for Java is entirely

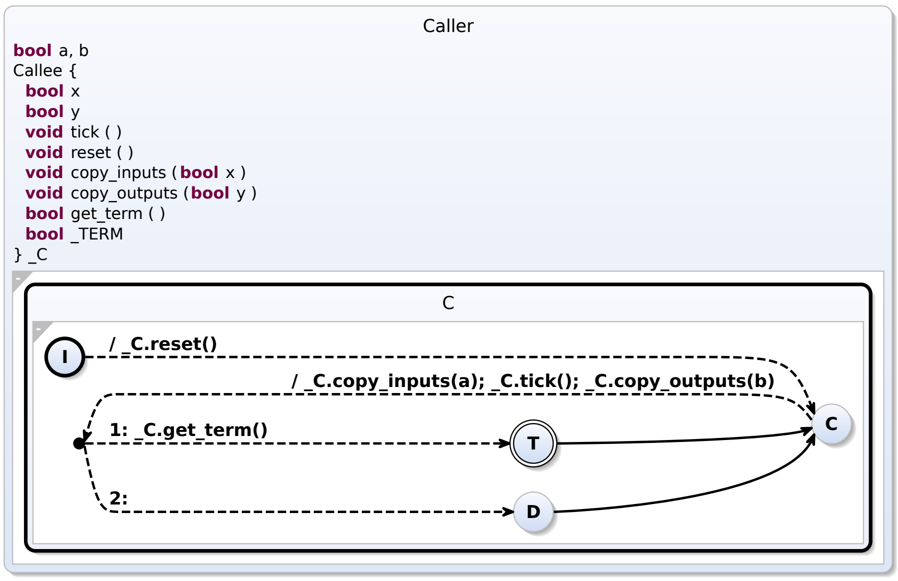


Figure 2.5. The model from Figure 2.2 after the alternative, non-final transformation

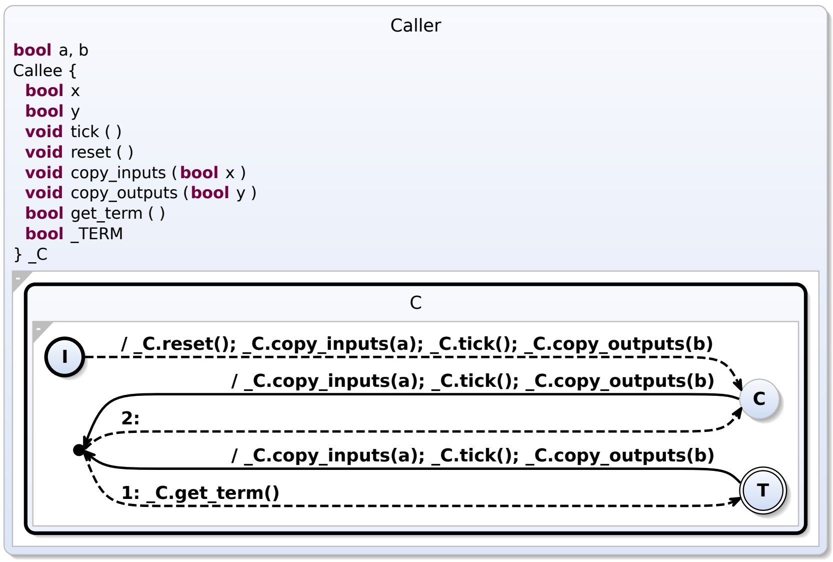
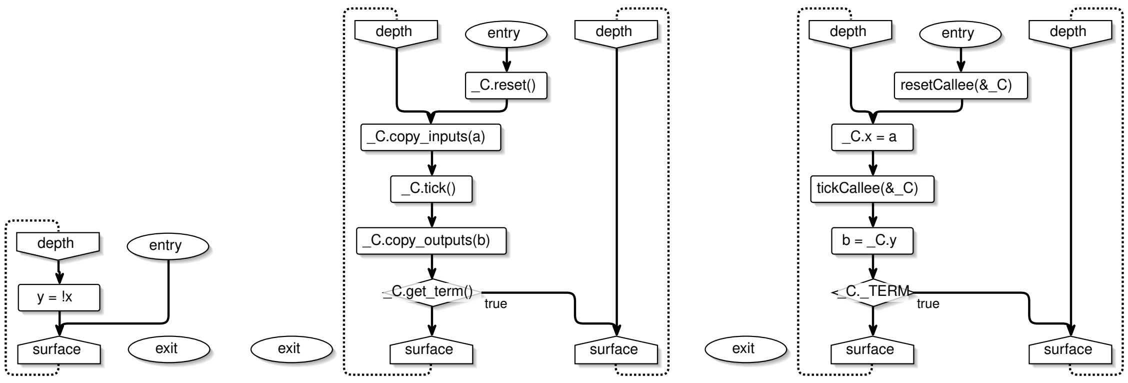

Caller
bool a, b
Callee {
bool x
bool y
void tick ( )
void reset ( )
void copy_inputs (bool x)
void copy_outputs (bool y)
bool get_term ( )
bool _TERM
} _C
C
I
/ _C.reset(); _C.copy_inputs(a); _C.tick(); _C.copy_outputs(b)
/ _C.copy_inputs(a); _C.tick(); _C.copy_outputs(b)
2:
/ _C.copy_inputs(a); _C.tick(); _C.copy_outputs(b)
1: _C.get_term()
C
T

Figure 2.6. The model from Figure 2.2 after the combined non-final and non-instantaneous transformation

analogous. Hence, this section describes only the C transformation in detail. As C does not have objects or methods, the syntax has to be modified accordingly. The transformation is as follows:

▻ The tick and reset method calls are replaced with function calls. The function's names are obtained by suffixing the method identifier with the callee's name. Its argument is a pointer to the callee instance.

▻ The get_term method call is replaced by a read access of the instance's termination flag.



(a) The callee SCG **(b)** The caller SCG before transformation **(c)** The caller SCG after transformation

Figure 2.7. The model from Figure 2.2 and Figure 2.3 before and after the module call SCG transformation

▻ The copy_inputs and copy_outputs method calls are replaced by a sequence of appropriate assignment operations.

Signals cannot be set to false once emitted. This enables a special optimization w.r.t. schedulability under the iur protocol: while outputs of any other type must be retrieved by an absolute write operation, signals are retrieved using a relative write, i. e., x |= Callee.y. Furthermore, input signals are not reset at the beginning of the tick—the caller or host software is expected to reset them. This allows a signal to be used as an input *and* output variable of a callee module. Otherwise, the fact that signals are initialized by absolute write in each tick would cause the initialization and retrieval of outputs to be in a write-write conflict.

The result of the transformation of the running example model is shown in Figure 2.7c. Note that the callee's SCG is not modified, provided that it does not itself have MCS references.

Figures 2.8 and 2.9 shows the code generated at the end of compilation. In Figure 2.8, it is visible that the callee's tick and reset functions, as well as its struct, are suffixed with the callee's name. Therefore, the definition of the TickData struct in Figure 2.9a and the function calls in Figure 2.9b are valid. The TickData struct contains a TickDataCallee struct. A pointer to this struct is handed into the callee's tick and reset functions. If the caller had multiple references to the callee, its TickData would simply contain multiple TickDataCallee instances. The implementation of the callee's tick and reset functions would not be replicated.

2.4 Separate and Simultaneous Compilation

There are two modes of compilation for MCS: *separate* and *simultaneous*. The simultaneous mode is similar to the way references are included when using MES. The callee module is present either within the same SCCharts source file as their caller or otherwise included from another SCCharts source file by an **import** statement. The models are then compiled side-by-side but independently from one another.

Separate compilation allows the inclusion of pre-compiled modules. This requires the generated target language files and an *SCCharts header file*. The header file is an SCCharts file with a special structure: it contains the name and input and output variables of the module and is annotated with **@header**, as well as any other annotations that apply to the module. In order to use a header file, the modeler simply includes it either in the caller's source code or by importing it. In order for a model to

```c
typedef struct {
        char x;
        char y;
        char _GO;
        char _TERM;
} TickDataCallee;

void resetCallee(TickDataCallee* d);
void logicCallee(TickDataCallee* d);
void tickCallee(TickDataCallee* d);
```

(a) Callee.h (abridged)

```c
#include "Callee.h"

void logicCallee(TickDataCallee* d) {
        if (!d->_GO) {
                d->x = !d->y;
        }
}

void resetCallee(TickDataCallee* d) {
        d->_GO = 1;
        d->_TERM = 0;
}

void tickCallee(TickDataCallee* d) {
        logicCallee(d);

        d->_GO = 0;
}
```

(b) Callee.c (abridged)

Figure 2.8. Callee code generated from the model introduced in Figure 2.2

be usable as a pre-compiled module, it must be prefaced with the `#code.naming suffix` pragma before compilation. Its tick and reset functions, as well as its TickData struct are then named as required.

Figure 2.10 shows an example module named Contains101, its SCCharts source code and an appropriate header file. The module is a simple state machine that accepts any sequence containing the subsequence [true, false, true]. Because it cannot terminate instantaneously, it is annotated with `@isDelayed`.

The SCCharts header file can be automatically generated. This includes detection of instantaneity and CFS. If a module is non-instantaneous, the `@isDelayed` annotation is added to the header. Similarly, if the module uses CFS, the header is annotated with `@nonFinal`. Note that the current implementation does not provide automatic generation of header files.

2.5 Parallels to Argument Handling

Many of the design decisions explained in this chapter have the explicit goal of making MCS as similar to MES as possible. That extends to syntax as well as semantics. There are, however, discrepancies roughly analogous to the difference between *call-by-reference* and *call-by-value* argument handling.

From a purely SCCharts perspective, i. e., before compilation, a modeler can view MES references as similar to a procedure in an imperative language that operates on pointers to its inputs and outputs. The comparison is somewhat misleading in the sense that conflicting concurrent writes to shared variables, a behavior normally associated with call-by-reference in a concurrent context, are precluded by the iur protocol. Conversely, it is accurate in the sense that the referenced module can arbitrarily write to input variables. Because these input variables are identical to the root model's variables they are bound to, these writes are persistent. Actually, MES does not differentiate between input and output variables at all. Every non-local variable is shared between the two models. Consequently, avoiding

```
#include "Callee.h"

typedef struct {
        char a;
        char b;
        TickDataCallee _C;
        char _g1;
        char _g5;
        char _GO;
        char _cg1;
        char _TERM;
        char _pg1;
} TickData;

void reset(TickData* d);
void logic(TickData* d);
void tick(TickData* d);
```

(a) Caller.h (abridged)

```
#include "Caller.h"

void logic(TickData* d) {
        d->_g5 = d->_pg1;
        d->_g5 = d->_GO || d->_g5;
        if (d->_g5) {
                d->_C.x = d->a;
                tickCallee(&d->_C);
                d->b = d->_C.y;
        }
        d->_cg1 = d->_C._TERM;
        d->_g1 = d->_g5 && !d->_cg1;
}

void reset(TickData* d) {
        d->_GO = 1;
        d->_TERM = 0;
        resetCallee(&d->_C);
        d->_pg1 = 0;
}

void tick(TickData* d) {
        logic(d);

        d->_pg1 = d->_g1;
        d->_GO = 0;
}
```

(b) Caller.c (abridged)

Figure 2.9. Caller code generated from the model introduced in Figure 2.2

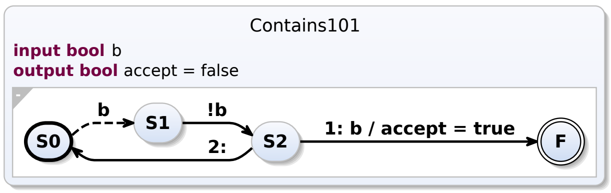
the modification of input variables, where undesired, is left to the modeler.

In the same sense, MCS is much more similar to functions using call-by-value. Writes to input variables inside the callee do not change the value of the caller's associated variable. Furthermore, only the callee can write to its own output variables. This departure from the MES interface variable handling is necessary in order to allow concurrent writes to input variables. If input variables were written back after execution of the tick function, write-write conflicts would arise.

2.6 Code Size and Performance Implications

The main advantage of MCS over MES is that it enables modular code generation. In addition to the benefits regarding unit tests, build times and closed-source distribution this provides, it is reasonable to assume MCS can also be superior to MES in terms of code size and performance.

If a single callee module is referenced multiple times, its tick function is defined only once. While multiple instances of a callee module each require a proxy state in the caller model, this state is likely much smaller and less complex than the module itself. Therefore, the generated code can be



(a) The Contains101 model

```
scchart Contains101 {
	input bool b
	output bool accept = false

	initial state S0
	immediate if b go to S1

	state S1
	if !b go to S2

	state S2
	if b do accept = true go to F
	go to S0

	final state F
}
```

(b) The source file Contains101.sctx

```
@header
@isDelayed
scchart Contains101 {
	input bool b
	output bool accept
}
```

(c) The header file Contains101-header.sctx

Figure 2.10. The Contains101 model along with its source and header file

conjectured to be smaller.

Even if the callee module is used only once, MCS has two further possible advantages. First, what would have been a single tick function using MES is split into multiple tick functions using MCS. These smaller functions may lead to more optimal caching behavior. Second, SCCharts compiled using the netlist-based synthesis recompute all registers and guards for each tick. If the callee's proxy state is not active in a tick, its tick function is not called. Therefore, such a tick is less computationally intensive overall. If multiple callees are active in at different times, but never in the same tick, worst-case execution time can be expected to decrease.

Chapter 3

Implementation

This chapter presents the implementation of MCS. Technologies used in the implementation are briefly described in Section 3.1 and the implementation itself is presented in Section 3.2.

3.1 Used Technologies

The implementation presented in this chapter is an addition to the KIELER project, specifically the KIELER Compiler (KiCo). KIELER integrates into the Eclipse IDE's Rich Client Platform (RCP).

3.1.1 Eclipse

Eclipse[1] is an open-source IDE originally released in 2004 as a tool for Java development that has since gained support for various programming languages and frameworks thanks to its extensibility: since version 3.0, Eclipse is made up of the Equinox framework[2], which serves as a plug-in host, and numerous plug-ins, collectively known as the Eclipse Rich Client Platform (RCP). This allows applications other than the Eclipse IDE itself to be built upon the platform and to use its dynamic plug-in management and extension points, as well as other features.

3.1.2 KIELER

One such application using the Eclipse RCP is KIELER. Figure 3.1 gives an overview of the KIELER project and its components. KIELER provides the reference implementation for SCCharts and serves as an IDE for SCCharts and many other synchronous languages, combining source editor, compiler, simulator, visualizer and debugger. An integral part of KIELER is KiCo, a modular compiler for synchronous languages. KiCo allows developers to specify individual operations on models as *processors* as described by Smyth et al. [SSH18].

[1] `https://www.eclipse.org/eclipseide/`
[2] `https://www.eclipse.org/equinox/`

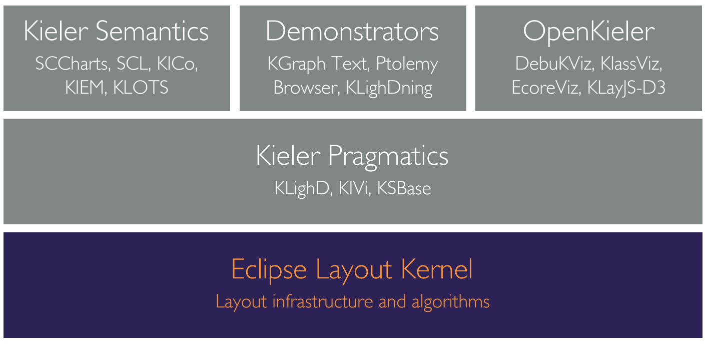

Kieler Semantics
SCCharts, SCL, KICo, KIEM, KLOTS
Demonstrators
KGraph Text, Ptolemy Browser, KLighDning
OpenKieler
DebuKViz, KlassViz, EcoreViz, KLayJS-D3
Kieler Pragmatics
KLighD, KIVi, KSBase
Eclipse Layout Kernel
Layout infrastructure and algorithms

Figure 3.1. The KIELER Project[3]

3.1.3 Xtend

Xtend[4] is a programming language layered on top of Java. It provides extension methods, in-line anonymous functions, type inference and many other conveniences. One of its features are Extension Methods, which allow developers to add methods to existing classes. Large parts of KiCo are implemented in Xtend.

3.2 Implementation into KIELER

KiCo compilation chains, or *systems*, are sequences of transformations. Each of these transformations is either itself a system or otherwise implemented as a processor. Most processors take a model and return a transformed version, although there are others. A system can be defined simply by listing the systems and processors in order of execution.

MCS is implemented in KIELER by two processors: `ReferenceCallPreprocessor`, which implements the MCS SCCharts transformation, and `ReferenceCallProcessor`, which implements the MCS SCG transformation. Both processors are implemented using Xtend.

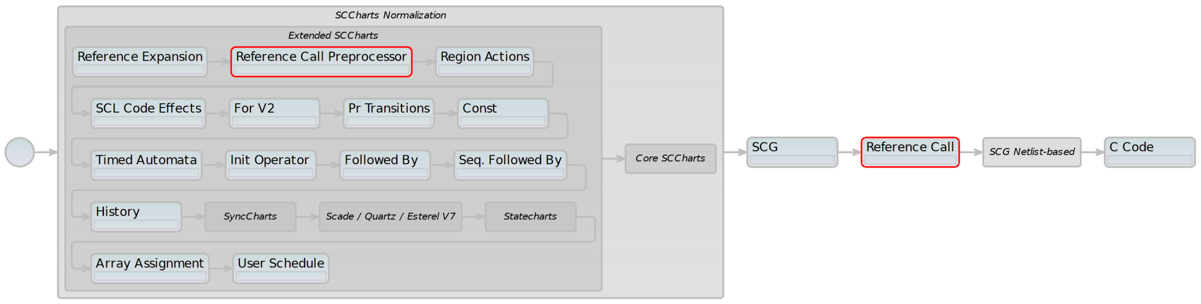


Figure 3.2. The modified netlist-based SCCharts to C compiler pipeline. Additions are highlighted in red.

Figure 3.2 shows the netlist-based C system with the newly added processors. The `ReferenceCallPreprocessor` is inserted into the Extended SCCharts system. It is executed directly

[3]`https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Overview`, accessed 12 September, 2021
[4]`https://www.eclipse.org/xtend/documentation/index.html`

after the module expansion processor, which implements MES. Since the `ReferenceCallPreprocessor` is the second processor in the system, it can use the entire Extended SCCharts feature set, excluding MES references.

The `ReferenceCallProcessor` is inserted into the netlist-based (C) system on the top level. A corresponding Java processor would take its place in the Java netlist-based system. It is executed after the model has been converted to SCG and optimized, but before the synthesis-specific (i. e., netlist-based or priority) transformations.

The primary design goal of both processors is readability and comprehensibility of intermediate models. KIELER allows the modeler to view—and even visualize the simulation of—every intermediate result of the compilation. This can help them debug and better understand their own model. This feature provides an incentive to SCCharts developers to encode even transient data into the transformed model instead of storing it externally.

3.2.1 The Reference Call Pre-processor

The class `ReferenceCallPreprocessor` in the package `de.cau.cs.kieler.sccharts.processors` implements the module call SCCharts transformation discussed in Section 2.3.1. A proxy class is created for each module. It acts as a stand-in for the callee's data structure. Recall that this data structure is defined outside of the caller and thus cannot be accessed during compilation. Hence, the stand-in is needed. Later, during C code generation, the proxy class is interpreted as a host type. Thus, all accesses to it are actually accesses to the callee module's data structure, defined in its source files. This is what allows separately compiled SCCharts modules to interoperate.

After the proxy class is created, a proxy state is created for each instance of a module. The proxy state is a proxy for the callee's behavior within the caller model. From the caller's perspective, the proxy state emulates the callee. It takes input values and returns the corresponding output values. If the callee terminates, so does the proxy state. This is achieved by accessing an instance of the proxy class. As the proxy class goes on to become a definition of the callee's data structure, its instances become actual instances of the data structure. Thus, all accesses to instance members are converted into accesses to the data structure's fields. This, in turn, is how actions in the proxy state can operate on callee variables that are defined in a separate source file. Table 3.1 summarizes the roles of significant methods of the `ReferenceCallPreprocessor`.

In the following, implementation considerations are illustrated on the example of the handling of bindings. Recall that in MES, the reference contains a set of binding definitions. A variable of the root SCChart is bound to an interface variable of the referenced model. In MES, this is accomplished by renaming variables during expansion. MCS uses the same syntax for binding definitions, albeit with call-by-value semantics. In MCS, values are copied from one variable to the other. This could have been accomplished by inserting assignment actions directly into the transitions of the proxy state. However, doing so could massively increase the amount of visual bloat in the intermediate model. This would directly violate the design goal of readability stated earlier in this section.

Thus, it was decided to keep the assignment of inputs and retrieval of outputs abstract by instead inserting calls to methods `copy_inputs` and `copy_outputs` defined in the proxy class. This has the additional benefit that future implementations could directly implement these methods without much re-engineering, e. g., for a more object-oriented Java implementation. However, the binding definition is stored as part of the reference, which is removed during the transformation. Hence, it must be stored elsewhere. As stated above, it is good practice to encode transient data within the model. Thus, it was decided to store the binding definition implicitly in both the parameters of the `copy_inputs` and `copy_outputs` methods and the arguments they are called with.

Table 3.1. Significant methods in `ReferenceCallPreprocessor`

`transform`	The entry point of the transformation. `transformRootState` is called for each root state. Subsequently, **`@header`** models are removed.
`transformRootState`	This method finds all module call references in a given model and transforms each as follows. Externally imported models are copied into the root model and recursed into. If no proxy class representing the model exists, it is created. Finally, the model's proxy state is created.
`createOrGetProxyClass`	This method returns the proxy class representing a given model. If none exists, it is first created according to the specification given in Section 2.3: The model's input and output variables and termination flag, as well as placeholder methods, are declared. The created class is annotated with **`@Model`** in order to differentiate it from user defined classes.
`createProxyState`	This method creates the proxy state for a module instance, delegating either to `createDelayedTransitions` or `createInstantaneousTransitions`, depending on whether or not the module is annotated with **`@isDelayed`**.
`createInstantaneousTransitions`	This method fills the module's proxy state with the instantaneous state machine shown in Figure 2.3. An extra transition is added if the module is annotated as **`@nonFinal`**; cf. Figure 2.5.
`createDelayedTransitions`	This method fills the module's proxy state with the non-instantaneous state machine shown in Figure 2.4. Similar to `createInstantaneousTransitions`, a transition is added if the module is **`@nonFinal`**.

The parameters of copy_inputs are the input variables of the callee module in order of declaration. These are known either from the callee header file or its source file. The arguments of the method call are the caller variables from the binding definition. Their order is determined by looking up each variable's bound callee variable and sorting them according to the order of declaration. Thus, their respective index in the argument list matches the index of their bound callee variable in the parameter list. Output variables are handled analogously.

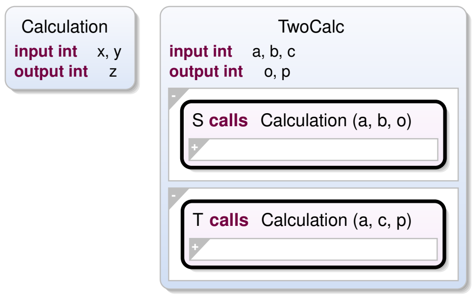


Figure 3.3. The TwoCalc model

Figure 3.3 shows an example model. The callee module Calculation has input variables x and y and an output variable z. The caller model TwoCalc has two instances of Calculation, one with bindings

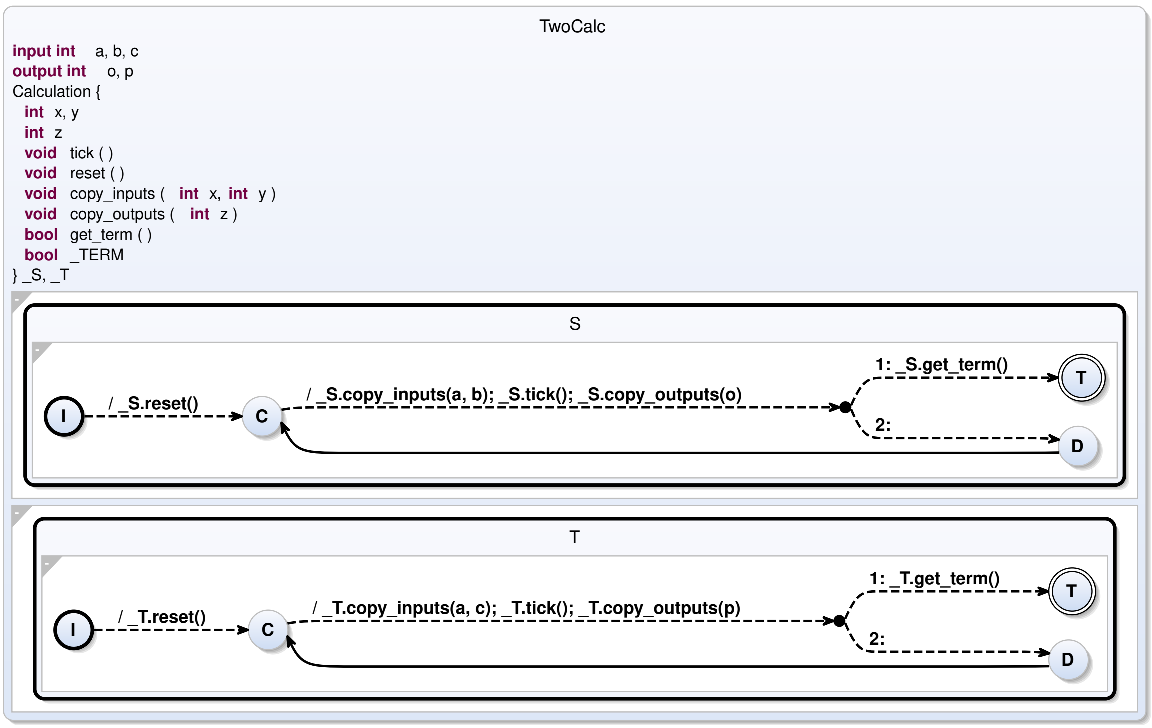


Figure 3.4. The TwoCalc model after transformation

(a, b, o) and one with bindings (a, c, p). Figure 3.4 shows how the bindings are encoded into the model. x and y become the parameters of copy_inputs, z the sole parameter of copy_outputs. In proxy state S, copy_inputs is called with arguments (a, b). This reflects that a is bound to x and b to y. In T, the arguments are (a, c), again reflecting the binding.

The method filteredParamsFromBindings shown in Figure 3.5 is the generic implementation used for both input and output parameters. Its parameter predicate determines whether input or output parameters are to be processed. Lines 7 through 9 perform the sorting of arguments according to their declaration order.

3.2.2 The Reference Call Processor

The class ReferenceCallProcessor in the package de.cau.cs.kieler.scg.processors implements the module call SCG transformation discussed in Section 2.3.2. Table 3.2 summarizes the methods defined in the class. The two tasks of this processor are the translation of syntax and the expansion of method calls into sequences of assignments.

Translating object-oriented syntax into C syntax is relatively straightforward in this case. Recall that callee data structure and function names are constructed by suffixing the callee's name to the structure or function name. Hence, a statement c.tick(), where c is an instance of Callee, is simply transformed into tickCallee(&c).

The expansion of method calls is more complex. For the copy_inputs method, a distinction is made between three cases:

(i) The method call has no arguments.

```
protected def List<Parameter> filteredParamsFromBindings(State ref, ValuedObject instance,
        Function1<Binding, Boolean> predicate) {
        val bindings = ref.createBindings
        val parameters = <Parameter>newArrayList
        val classVarVOs = ref.classDeclaration.declarations.filter(VariableDeclaration)
                .map[valuedObjects].flatten.toList
        for (binding : bindings.filter(predicate).sortBy [ b |
                classVarVOs.indexOf(classVarVOs.findFirst[name == b.targetValuedObject.name])
        ]) {
                parameters.add(createParameter => [
                        expression = binding.sourceExpression.copy
                ])
        }

        return parameters
}
```

Figure 3.5. The `filteredParamsFromBindings` method

Table 3.2. Methods in `ReferenceCallProcessor`

`transformAll`	The entry point of the transformation. Calls `transform` for each SCG.
`transform`	This method is the implementation of the transformation discussed in Section 2.3. All tick and reset method calls are replaced by function calls. The copy_inputs and copy_outputs method calls are each replaced with a sequence of assignment nodes. Calls to get_term, all of which are contained within conditional nodes, are each replaced with a reference to their callee's termination flag.
`getModuleClasses`	This method finds all classes annotated with **`@Module`**. It is invoked by `transform` in order to guard user-defined classes and method calls against inadvertent modification.
`handleOperatorExpression`	This method recursively inspects an operator expression such as a logical conjunction for calls to get_term and replaces them appropriately.

(ii) The method call has exactly one argument.

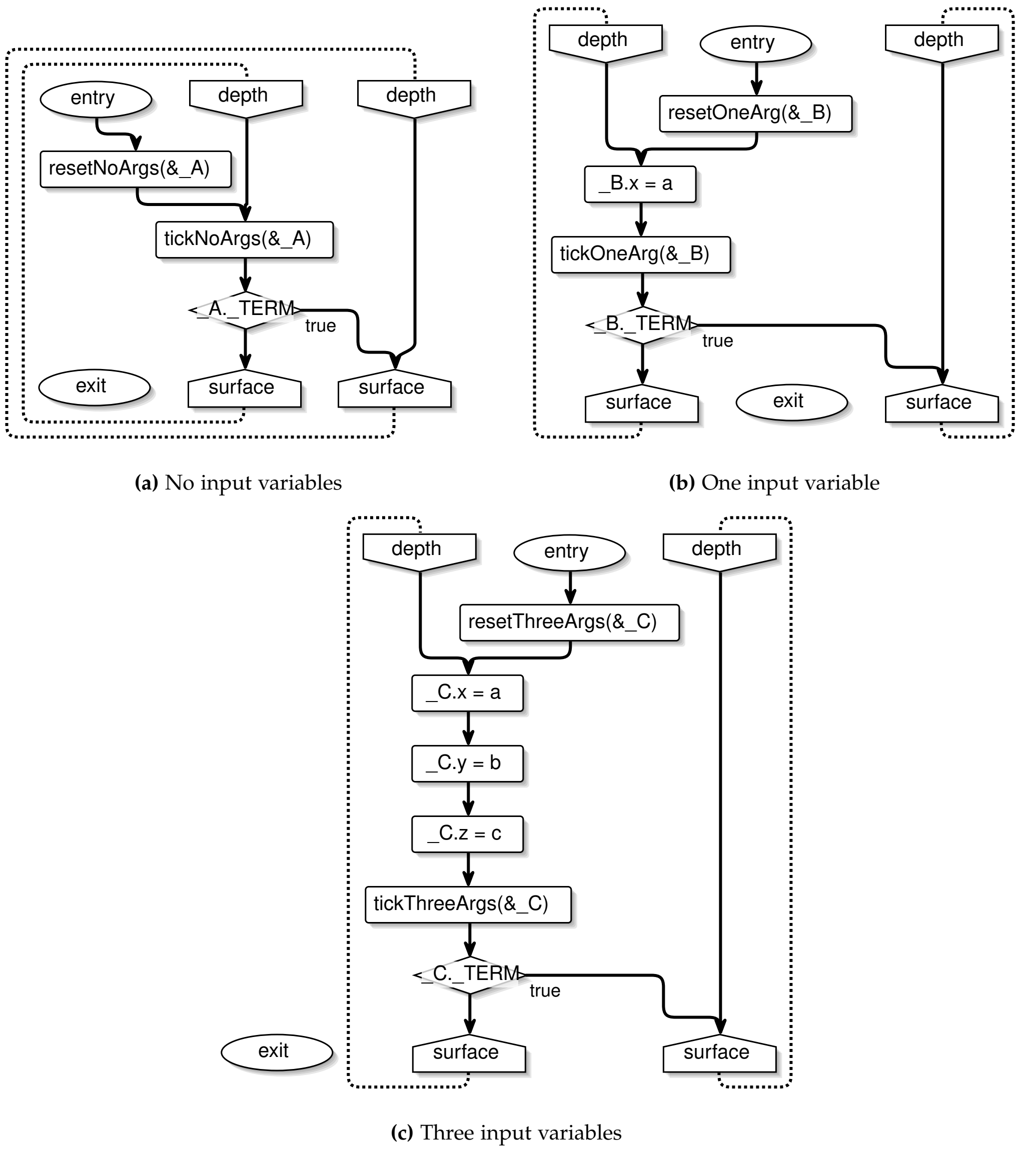
(iii) The method call has more than one argument.

(i) corresponds to a callee that does not have any inputs. In this case, the node is skipped entirely. All incoming edges of the node are redirected to its target and the node is removed from the SCG.

(ii) corresponds to exactly one input being copied to the callee. In this case, the topology of the SCG can remain unchanged. The statement of the node containing the call is simply replaced by an assignment operation.

(iii) corresponds to multiple input variables and is handled similarly. The original node is replaced by the assignment of the first parameter. All subsequent assignments are placed in newly created nodes. These assignment nodes are then chained by pointing the outgoing edge of the original node to the second node, the second to the third, etc. The outgoing edge of the last node is then pointed to the target of the original node. Figure 3.6 shows the caller SCG for a callee with zero, one or three input variables and no output variables.

Calls to copy_outputs are handled analogously. Finally, calls to get_term are replaced with the callee's termination flag. Generally, the procedure is similar to case (ii) above. However, SCG-level optimization can introduce additional complexity. For instance, two chained conditional nodes can be combined into one conditional node. The condition of this new node is the logical conjunction of the two nodes' conditions. Hence, if the condition of a conditional node is a complex expression, it must be recursively searched for calls to get_term.



(a) No input variables

(b) One input variable

(c) Three input variables

Figure 3.6. Caller SCG for different numbers of callee input variables

Chapter 4

Evaluation

This chapter evaluates both the concept of MCS and the approach taken in this thesis, demonstrating their theoretical and practical limitations and advantages. As a central part of this chapter, performance measurements on several models, both synthetic and real-world, are given and analyzed. From this, recommendations for modelers are derived.

4.1 Supported Model Elements

There are some SCCharts model elements that cannot be supported by MCS for structural reasons. Others could not be implemented within the time constraints of this thesis. This section breaks down the model elements that SCCharts provides w.r.t. MCS's ability to support them. The results of this section are based on manual tests.

4.1.1 Types

Callee modules support all SCCharts types for local variables without any constraints. For interface variables, i. e., variables declared `input` and/or `output`, certain conditions apply.

Base types. The following base types can be used as input and output variables without any restrictions: `bool`, `int`, `float` and `signal`. As `string` is an alias for a `char*` in C, caution is advised. While a model using strings for caller-callee communication will compile, there is no protection against buffer overflows, etc. Therefore, it is not recommended to use strings.

`host` types are generally supported, provided they support assignment. E. g., if a `host` type aliases `string` or any other array type, it is unsupported; if it is a `struct`, regular `struct` assignment[1] is used.

Arrays. Structurally, arrays cannot be used as input or output variables at all. The reason is that C does not support assignments to arrays. While there are two obvious solutions, each of them carries heavy disadvantages: it is possible, instead of using the assignment operator to copy an array, to insert a call to the appropriate auxiliary copy function (i. e., `strcpy` or `memcpy`). Doing so would cause every array to be copied each time a callee model that declares it as an input or output variable is called, leading to potentially severe performance penalties. The alternative solution is to use a pointer to the array. However, this would be inconsistent with the call-by-value philosophy discussed in Section 2.5.

Constants and static variables. While local constants can be used—and behave as expected—in callee modules, `const input` and `const output` variables are currently not supported. A local `const` variable a in the caller model can be bound to a (non-constant) input variable b of the callee, in which case b is assigned the constant value of a each time the callee is called.

[1] struct members are shallow-copied

`static` variables in SCCharts are similar to static variables in C, i. e., they are not reset on state re-entry as described by von Hanxleden et al. [HDM+13]. Because `static` variables are currently reset in the first tick of every root model, the `static` keyword does not affect variables whose scope is the root level of a callee module. Until this is fixed, a possible workaround is wrapping the entirety of the referenced model inside a complex state.

4.1.2 Actions, Hierarchy and Concurrency

In callee models, all kinds of actions, hierarchy and concurrency except for CFSs can be used without any additional restrictions. As discussed in Section 2.3.1, CFSs have to be enabled manually by annotating the callee with `@nonFinal`, leading to the callee's tick function being called after termination.

However, the CFS transformation does not yet properly support multiple root SCCharts and erroneously inserts accesses to variables local to the callee into the caller model, leading to rejection by the C compiler. There is a workaround: if the callee is compiled separately from the caller instead of simultaneously (cf. Section 2.4), the erroneous accesses described above do not occur. In doing so, CFS can be used.

In caller models, module call references can be placed within any state or region; cf. Section 4.2 for the ensuing scheduling constraints.

4.1.3 Suspend and Complex Transitions

Suspend. Normal (strong) `suspend`, which specifies that a complex state is inactive while a condition is met, is fully supported; irrespective of whether suspension is used within a referenced model or in a complex state containing a module call reference, the behavior is identical to a module expansion reference in an otherwise identical setting. While `weak suspend` should conceptually behave as expected, it is still regarded as an experimental feature in SCCharts, therefore no correctness guarantees can be given when used in conjunction with MCS.

Abort. Both weak and strong `abort` transitions function as intended. Of course, if the condition of a strong `abort` is (or depends on) an output variable of the referenced model, no valid schedule exists. However, this limitation is not exclusive to MCS.

History. Deep `history` transitions are inherently supported because *skipping* the proxy state's initial state causes the callee's reset function to not be called. `shallow history` transitions, on the other hand, do not function as expected. If a referencing state is transitioned into using a `shallow history` transition, it behaves exactly as if a deep `history` transition had been used.

Join. Due to the proxy state's mirroring of the referenced model's termination status, outgoing `join` transitions from module call references function correctly.

Deferred. As `deferred` transitions are still experimental in SCCharts, no guarantees regarding their correct handling with MCS can be given. However, there are no known issues.

4.1.4 Mixed References.

MCS and MES references can be used within the same model. However, not all configurations are supported at the time of writing. If the references occur in parallel, i. e., an MCS and an MES reference in the same model before compilation, the references are resolved correctly.

If references are nested, issues arise: an MES reference inside a model referenced by MCS is not expanded because module expansion occurs before the MCS processing; an MCS reference inside an MES reference is erroneously expanded by the MES processor. The former issue can be worked around by separately compiling the referenced model. On their own, MCS references can be nested to arbitrary levels of depth.

4.2 Limitations

In addition to the restrictions imposed by the current implementation and discussed in the previous section, there is a conceptual limitation to the applicability of MCS. Some valid SCCharts are rejected by the compiler if MCS references are substituted for their MES counterparts.

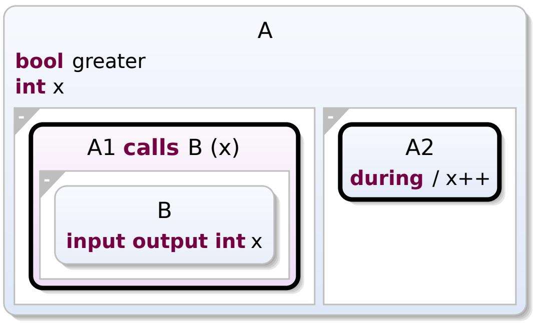


Figure 4.1. A minimal model not schedulable using MCS

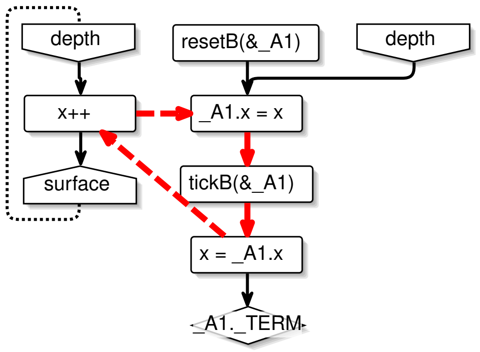


Figure 4.2. The dependency cycle introduced by MCS

The prime limitation of MCS is due to the design decision of subscribing to a call-by-value-like binding resolution. This decision leads to assignment statements in each tick before and after the callee's tick function is called. Excluding signals, these are always absolute write operations. Therefore, any model that introduces additional dependencies between one of the referenced model's outputs and one of its inputs is not schedulable according to the iur protocol, irrespective of the callee's internal behavior. Even if operations are entirely confluent, the callee model is a black box to the compiler. Figures 4.1 and 4.2 show a minimal example of a model affected by this.

Because all inputs are read before the callee model's tick function is called and all outputs are written after it has returned, if the caller model requires an output of a reference in order to determine an input—back-and-forth communication in other words—MCS does not support it. This pattern can be found in many larger SCCharts models, for instance in mutual exclusion: if a model requires a lock, it can only request it by setting an output variable. It can only receive the lock by reading an input value. Therefore, at least one tick boundary is required to obtain a lock. In other words, MCS prohibits instantaneous mutual exclusion.

One model that is impacted by this is the *Railway Controller*, the original motivation for referenced SCCharts [SMS+15]. Consequently, the model, which is part of the motivation for MCS, cannot be used to gauge its performance. It would be worthwhile to investigate whether this design limitation can be overcome without sacrificing functionality. The resulting model could also serve as a large-scale comparative benchmark.

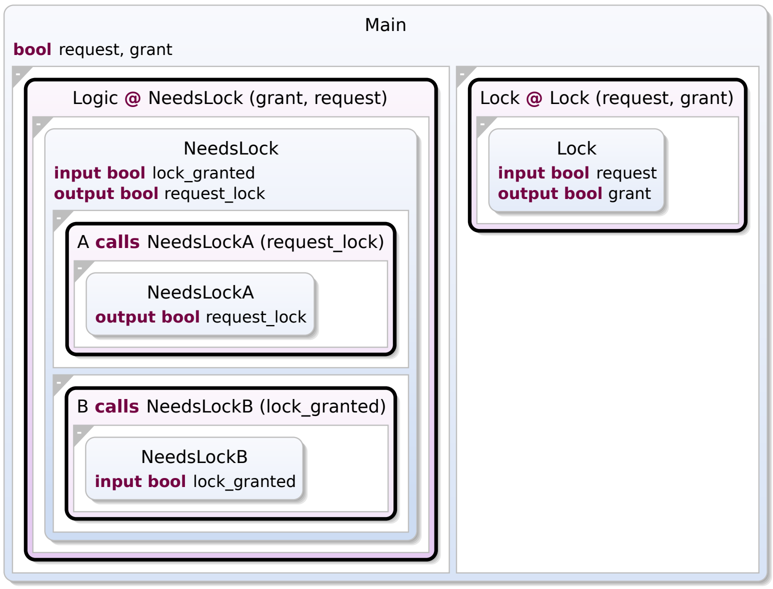


Figure 4.3. A sketch of a workaround for the mutex problem

A possible workaround for the mutex problem is shown in Figure 4.3. A module is manually split into two parts such that the first module, `NeedLockA`, computes whether a lock is needed and only the second module, `NeedLockB`, actually needs the lock. The two modules are then placed as MCS references within the `NeedLock` module. This module can then be included using an MES reference. The MES reference allows the caller to execute instructions between the calls to `NeedsLockA` and `NeedsLockB`. This solution places two proxy states in the caller model and therefore needs twice the space of a potential single-reference solution. However, the end result can still be smaller than not using MCS at all, provided the module is large.

Models that rely on the characteristic of MES that causes write operations to a module's *input* variables to propagate back to the caller model as discussed in Section 2.5 also diverge when MCS is used instead. The cause is the design decision to only copy the values of *output* variables back to the caller.

4.3 Performance Analysis

Section 2.2 introduces the hypothesis that MCS can, if proxy state induced overhead is less significant than size savings and gained optimizability, improve the performance of SCCharts that make use of references in terms of computation time per tick, as well as generated code and executable size. This section describes the steps undertaken in order to evaluate this hypothesis.

4.3.1 Methodology

The performance of a compiled SCCharts model, like any program, depends on many variables. The size of the model, its hierarchical structure, the synthesis used, as well as the back-end compiler's configuration play an important role. Furthermore, it cannot be assumed that statistical computation time for every input and every internal state is equal. This is especially doubtful in the case of MCS because entire sections of code are not executed in a tick in which a callee's tick function is not called. Therefore, a number of A/B benchmark tests must be conducted on multiple models with varying structural characteristics.

In order to ensure consistency, these models must also be supplied with deterministic inputs for each tick. KIELER supports such a functionality in the form of traces. When simulating a model, inputs and outputs can be captured and written to a trace file. This trace file can be read during subsequent simulations, with the added benefit that deviations between a model's output and a pre-recorded trace can be automatically detected and flagged. Therefore, if any semantic differences between MCS and MES were to arise in the tested models, they would not remain undetected.

The KiCo Command Line Interface (CLI) features a highly configurable benchmark tool alongside its automated testing framework. Given a list of models, each of which requires a trace file, the tool embeds each compiled model in a simulation executable that sets its inputs, executes its tick function and measures the *wall clock* execution time for each tick in the associated trace. Additionally, the Lines of Code (LoC) of the generated source code is determined as a secondary statistic.

The measurements in this section were obtained using the above-mentioned tool. For every trace and both reference resolution approaches, 50 runs were analyzed in order to be able to effectively identify and remove outliers, i. e., individual ticks during which external influences like kernel interrupts, etc. artificially increase the measured execution time. Furthermore, this allows for a closer examination of the distribution of tick times.

All models were compiled using the netlist-based synthesis using the C back-end, as it is currently the only one that supports MCS. Unless otherwise specified, GCC was configured with the `-O3` optimization level[2] and all tests were conducted on an Intel® Xeon® E5540 CPU with a base clock frequency of 2.53 GHz.

Outlier detection, where applied, was conducted as described by the following pseudocode:

```
for each trace and approach:
        for each tick:
                calculate lower and upper quartiles LQ, UQ of tick times across all runs
        for each tick and run:
                if time > UQ + 3 * (UQ-LQ):
                        remove the measurement
```

[2]`https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`, accessed 3 September 2021

This is a version of Interquartile Range (IQR) based outlier detection as described in [RH18], applied across runs, but only to one tick at a time.

4.3.2 Evaluated Models

Suitable SCCharts models were extracted from the Real-Time and Embedded Systems Group's model repository and semantically equivalent MCS versions were created by substituting each module expansion reference with an appropriate module call reference. Suitability, in this context, is determined by:

(i) usage of referenced SCCharts;

(ii) applicability to real-world tasks; and

(iii) schedulability of the model after module call reference substitution.

Most models in the repository are either structural tests and are therefore not complex enough to represent models that solve real-world tasks or otherwise do not use referenced SCCharts. Many of those that do were not designed with MCS in mind and would require a total refactoring in order to be schedulable.

However, two archived student homework submissions fulfill all three requirements. These models are `Backhoe` (Figure 4.4), a hypothetical excavator controller, and `Barcode` (Figure 4.5), a model that reads and validates simplified EAN-style bar codes [SMS+19]. These are not pictured here because no declaration of consent regarding publication could be obtained from the authors. In their stead, diagrams of their relevant structural elements have been created. Furthermore, no real-world execution traces for these models exist. It is, however, not difficult to infer their intended use and synthesize plausible traces.

Additionally, three synthetic highly modular models were created in order to provide a best case scenario for MCS. They are designed to feature multiple references to the same SCCharts model as well as a deep reference hierarchy in order to achieve optimal code size and caching behavior for MCS over MES. It would, of course, be preferable to use existing real-world models for this analysis. However, because of the communication restriction described in Section 4.2, most existing larger models cannot be scheduled using MCS. It is conjectured that many of these could be modified in such a way that they could be scheduled without making them significantly more computationally expensive, but doing so is outside the scope of this thesis.

Summary of all models. Table 4.1 summarizes the number of unique and overall references for each model. Multiple orders of magnitude, both of absolute number and of ratio between overall and unique references[3] are represented.

Table 4.1. References and unique references.

Model	Barcode	Backhoe	Counter (half width)	Counter (full frequency)	Counter
#References	1	2	19	32	35
#Unique	1	2	6	5	6

[3]References to different models; if a model has two references to one and the same model, it is interpreted to have two references and one unique reference.

The Backhoe model features two mutually exclusive references, each of which does not itself contain any references. During run-time, control flow cycles between the two, such that in almost every tick, one of the two will be active.

In a real-world scenario, operation generally follows this pattern: first, a button is pushed in order to initiate a process, e. g., extending the stick, boom and bucket assembly. The model then sets the actuator for extending the stick and awaits a signal from the relevant endstop sensor. This process is repeated for the boom and the stick. Each phase involves setting an actuator and awaiting feedback from a sensor, indicating the end of the phase; a common pattern in embedded systems.

Therefore, the model can operate either periodically, computing its reaction once per fixed time interval, or dynamically, reacting only whenever any input changes. In order to account for both use cases, as well as investigate whether one approach significantly benefits from one of these modes of operation, two separate traces were created: the *dense* trace, representing dynamic operation by varying inputs between each tick, and the *sparse* trace, which represents periodical operation by interspersing each pair of ticks from the dense trace with 100 ticks during which the inputs do not change.

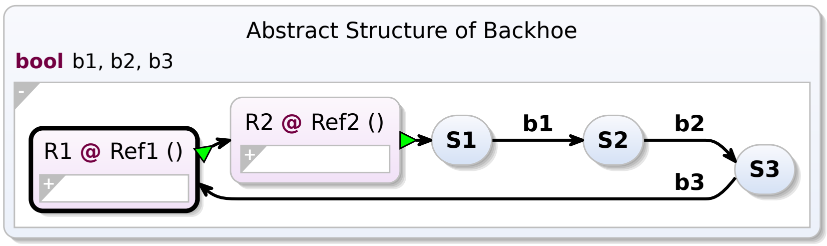


Figure 4.4. An abstract structural diagram of the Backhoe model

The Barcode model features only one reference that contains the vast majority of its model elements. It serves to investigate the overhead that MCS introduce in a worst-case scenario. The use case for this model is to feed it the segments of a bar code in a boolean format—light or dark—in successive ticks. The encoded number is calculated and validated against a checksum digit.

Because each tick corresponds to exactly one segment of the bar code, only a dense trace consisting of the model reading a bar code representing the number 42 was created for this model.

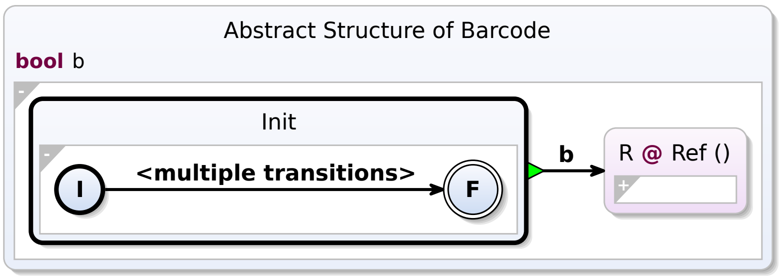
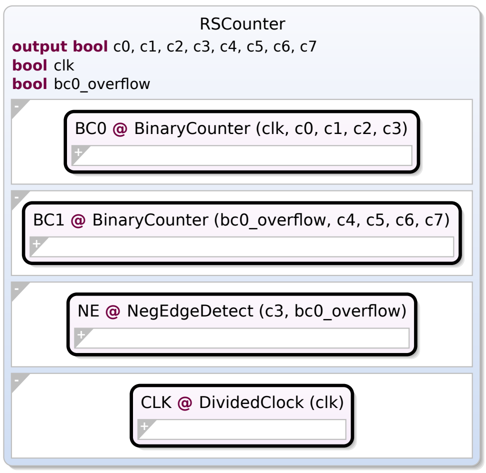
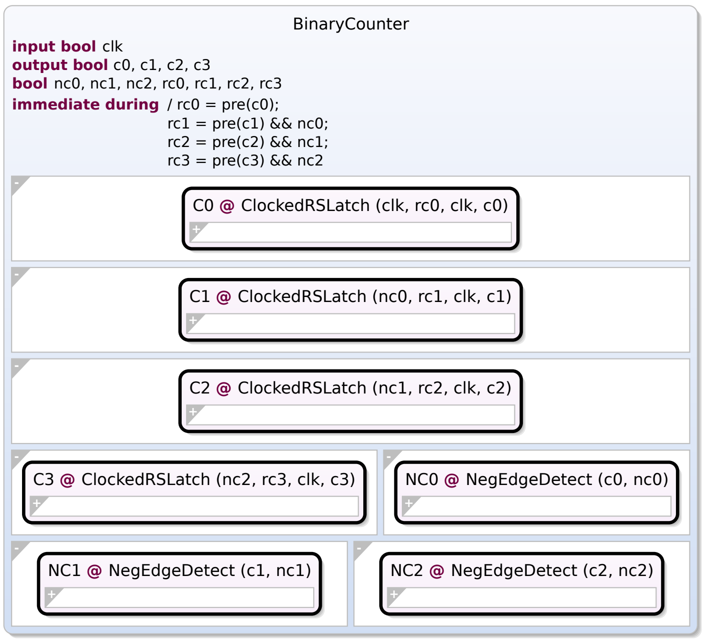


Figure 4.5. An abstract structural diagram of the Barcode model

The Counter model (Figure 4.6) is an 8-bit binary counter that counts every eighth tick. It is made up of two 4-bit binary counters, the overflow of the first of which, as detected by a negative edge detector, serves as a clock for the second. Each binary counter, in turn, uses clocked RS latches that


RSCounter
output bool c0, c1, c2, c3, c4, c5, c6, c7
bool clk
bool bc0_overflow
BC0 @ BinaryCounter (clk, c0, c1, c2, c3)
BC1 @ BinaryCounter (bc0_overflow, c4, c5, c6, c7)
NE @ NegEdgeDetect (c3, bc0_overflow)
CLK @ DividedClock (clk)

(a) The main model

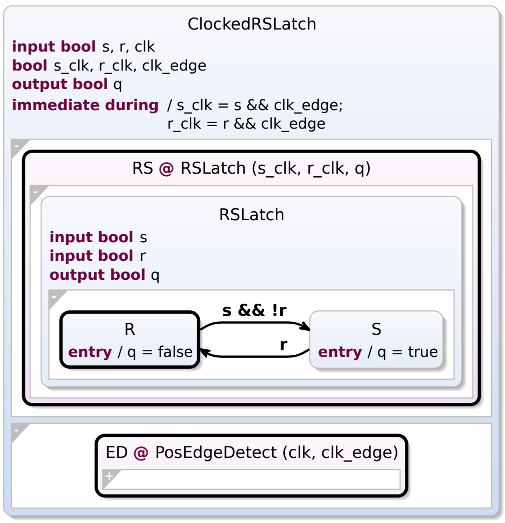

BinaryCounter
input bool clk
output bool c0, c1, c2, c3
bool nc0, nc1, nc2, rc0, rc1, rc2, rc3
immediate during / rc0 = pre(c0);
rc1 = pre(c1) && nc0;
rc2 = pre(c2) && nc1;
rc3 = pre(c3) && nc2
C0 @ ClockedRSLatch (clk, rc0, clk, c0)
C1 @ ClockedRSLatch (nc0, rc1, clk, c1)
C2 @ ClockedRSLatch (nc1, rc2, clk, c2)
C3 @ ClockedRSLatch (nc2, rc3, clk, c3)
NC0 @ NegEdgeDetect (c0, nc0)
NC1 @ NegEdgeDetect (c1, nc1)
NC2 @ NegEdgeDetect (c2, nc2)

(b) The half width binary counter module

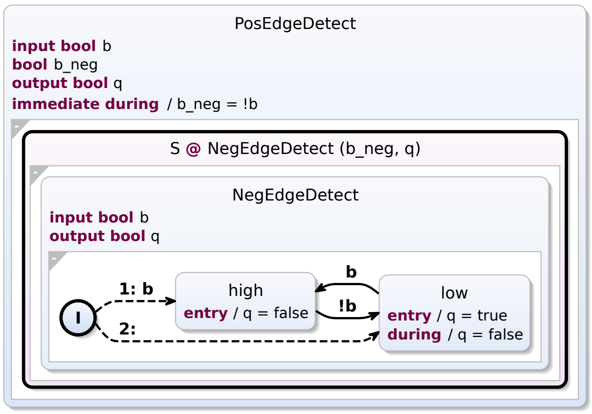
Figure 4.6. The Counter model and its sub-modules

are themselves made up of positive edge detection and an RS latch. As Figure 4.7 shows, the model, despite consisting of very simple modules, is quite large and complex after module expansion.

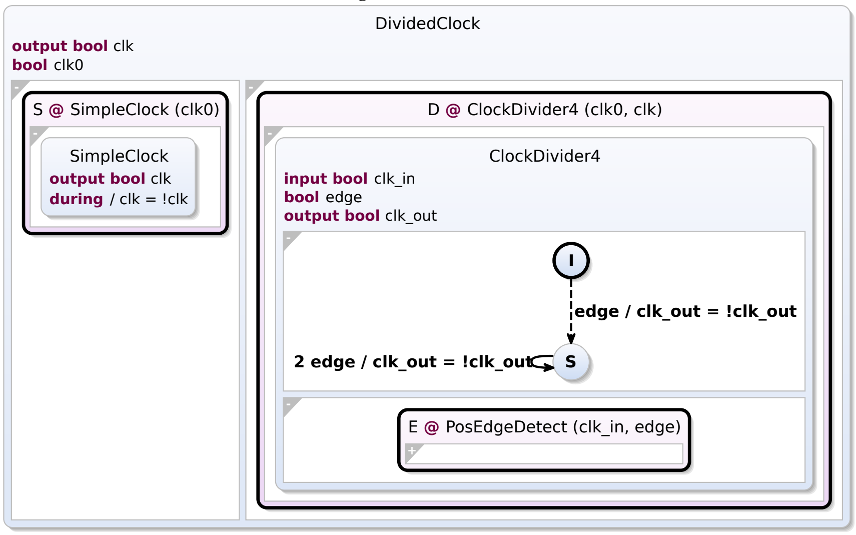
The Counter (full frequency) model. The clock divider was originally intended only as a means to add complexity to the model, but it gives an additional opportunity to analyze whether internal state changes have a significant computational cost by also testing a variant without a clock divider. Figure 4.8a shows that modified model. It is slightly smaller due to the missing clock divider, but its internal state changes in every tick.



(c) The clocked RS latch module



(d) Edge detection modules



(e) The divided clock module

Figure 4.6. The Counter model and its sub-modules (cont.)

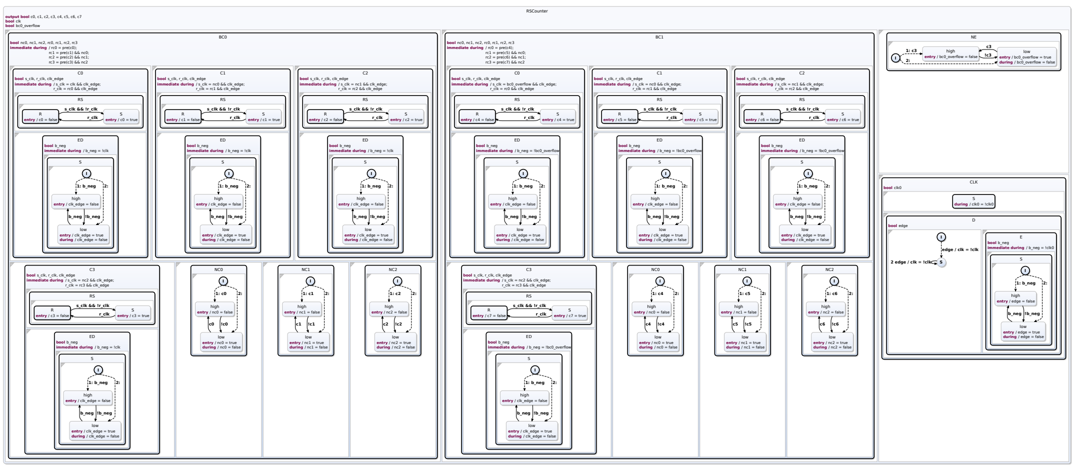
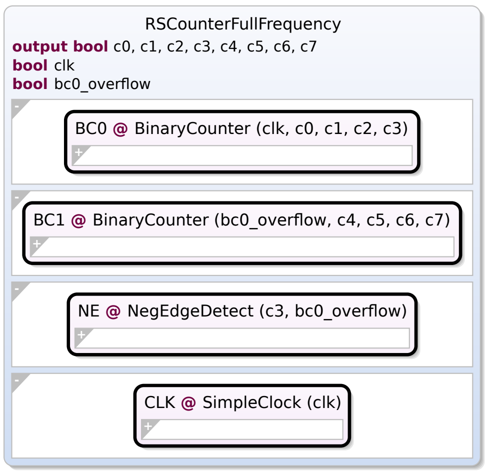
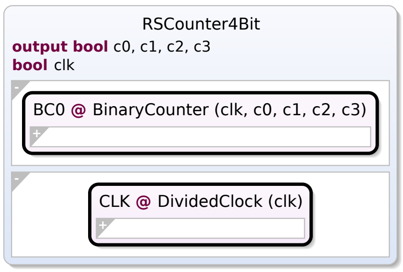


Figure 4.7. The Counter model after module expansion

The Counter (half width) model. The Counter model also enables a controlled test for code size and performance behavior in cases where additional references to already referenced models are introduced. The half width variant (cf. Figure 4.8b) was constructed for this purpose. It uses only one 4-bit binary counter and is thus about half the size of the original model after module expansion, but has the same unique references.


RSCounterFullFrequency
output bool c0, c1, c2, c3, c4, c5, c6, c7
bool clk
bool bc0_overflow
BC0 @ BinaryCounter (clk, c0, c1, c2, c3)
BC1 @ BinaryCounter (bc0_overflow, c4, c5, c6, c7)
NE @ NegEdgeDetect (c3, bc0_overflow)
CLK @ SimpleClock (clk)

(a) The Counter (full frequency) model


RSCounter4Bit
output bool c0, c1, c2, c3
bool clk
BC0 @ BinaryCounter (clk, c0, c1, c2, c3)
CLK @ DividedClock (clk)

(b) The Counter (half width) model

Figure 4.8. Counter variations

4.3.3 Analysis

Figure 4.9 shows the mean and standard deviation of the tick time for each trace and approach after removing outliers. Models are ordered from left to right by ascending number of references. It is immediately apparent that the overhead introduced by MCS does not impact mean performance by a large amount. The opposite seems to be true: in most cases, MCS is significantly faster than MES.

Barcode. The only model that does not significantly benefit from the approach is Barcode. This is easily explained by the fact that it features only one reference that is active for most of the trace. Figure 4.10a plots the distribution of the central 50 percent of time for each tick and shows that while MCS is at least four times faster in the initial and final few ticks, during which the reference is not active, the mid section is both on average faster when using MES as well as having much less variance.

For convenient comparison, this interval (all ticks but the first and last six) is also plotted in Figure 4.9. The difference between the whole trace and its mid section is evidence that MCS can profit from references that are inactive for large intervals of time, but that the additional overhead that occurs when it is active must not be overlooked.

Figure 4.10b is a histogram of the measured tick times plotted against their frequency of occurrence over all ticks and all runs; it paints a similar picture by illustrating the distribution of tick times. There is a clearly visible cluster of sub-millisecond ticks for MCS representing the ticks during which the reference is inactive.

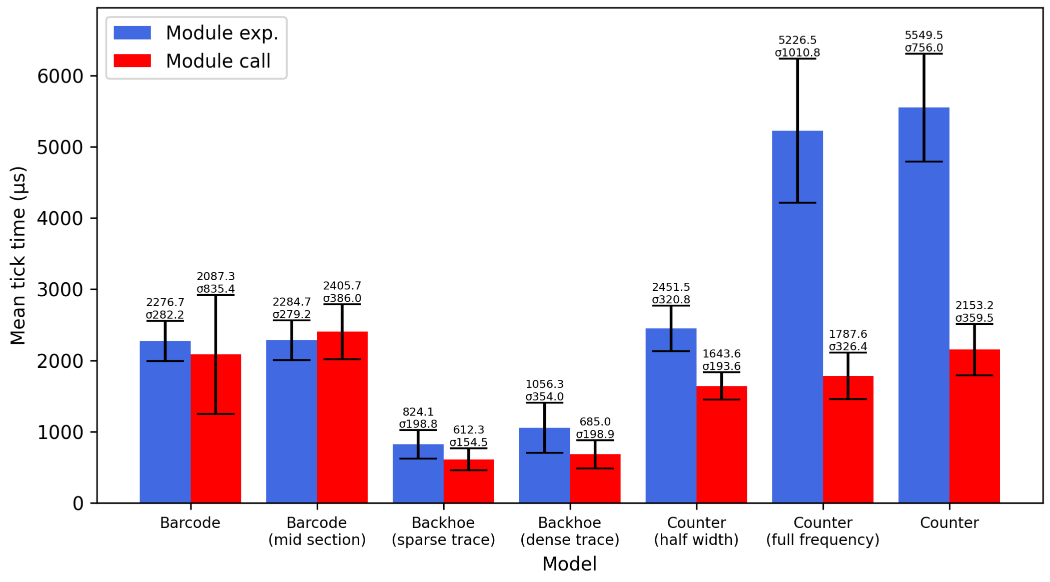


Figure 4.9. Mean aggregated tick times and tick time standard deviation for all traces. Models are ordered from left to right by ascending number of references.

Backhoe. The Backhoe model significantly benefits from MCS, improving the mean tick time by 35% and 26% in the dense and sparse cases, respectively. The distribution of these numbers seems to suggest that MCS can profit more from denser traces, i. e., models that get called dynamically and only when inputs change.

However, this distribution can be partially explained by the effect of the (computationally expensive) initial tick on the mean. While the first tick makes up $\frac{1}{15}$ of the dense trace and thus can significantly influence the mean, it only makes up less than $\frac{1}{1000}$ of the sparse trace and thus the mean is entirely dominated by the statistical weight of the other ticks. In contrast, MCS has an initial tick for each reference, so its *initial tick penalty* is spread out over multiple ticks.

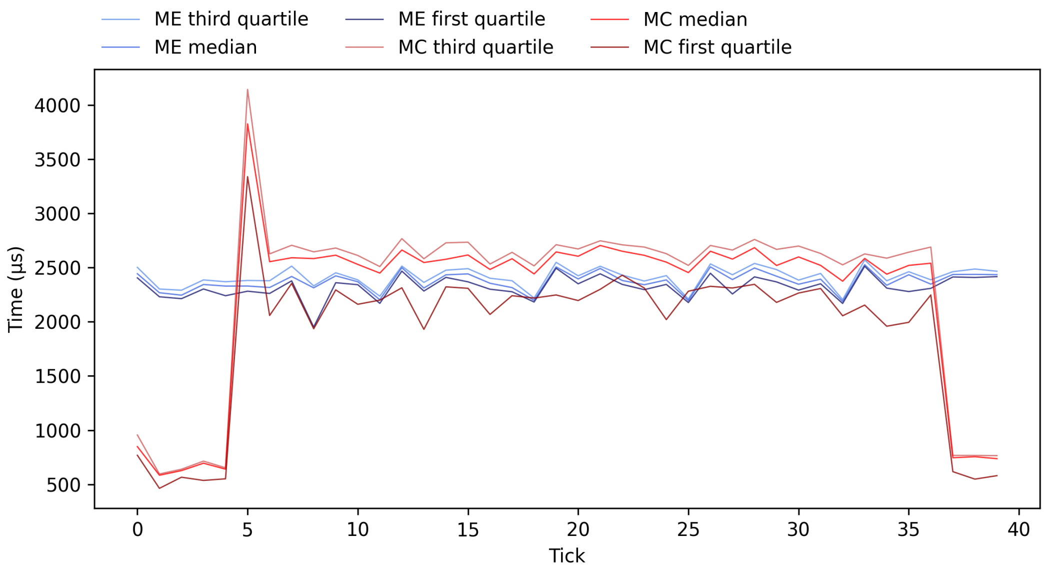
Figure 4.11 shows the tick time quartiles for both traces. Two effects become immediately apparent, especially in the sparse trace (Figure 4.11b):

(i) While the upper quartile and median are always close together, the lower quartile is significantly lower, especially for MES.

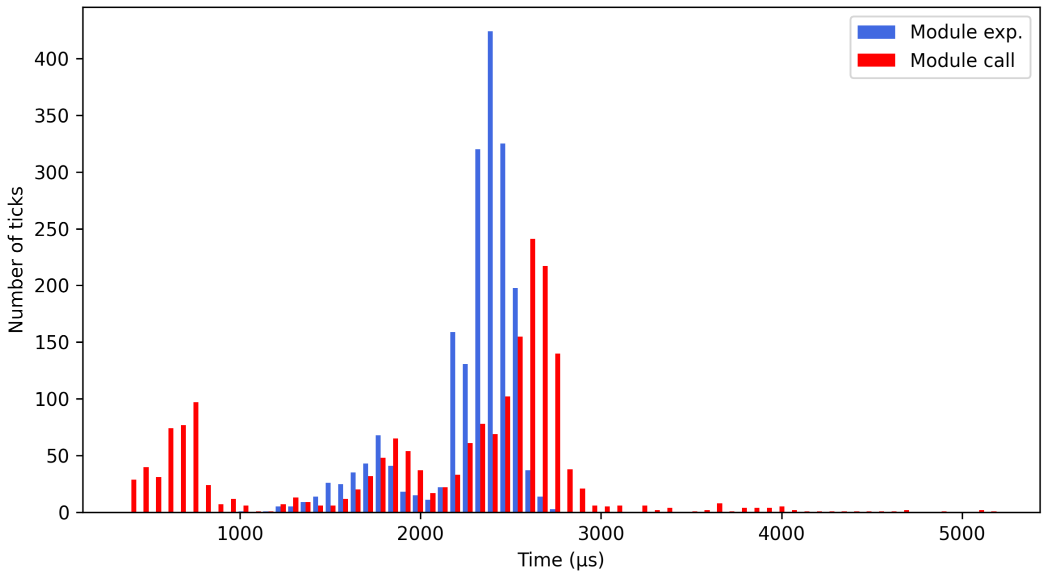
(ii) Between ticks 500 and 700 (resp. ticks 9 and 10 in the dense trace), MCS is substantially less computationally expensive.

While (ii) is easily explained by the fact that for exactly two transitions, no reference is active (cf. Figure 4.4) and thus their tick functions aren't called, (i) is more puzzling. In an ideal execution environment, the time taken to compute one and the same tick should be identical across runs. However, even when accounting for occasional long interruptions by the operating system, one would expect the upper quartile to diverge from the median, not the lower quartile.

There are at least two possible explanations for this phenomenon: either the low tick times of the first quartile are a product of fortunate, non-deterministic processor-level effects, e. g., speculative execution, favorable caching, etc., or the operating system interrupts execution for a duration of 200 to



(a) Median and upper and lower quartiles for each tick

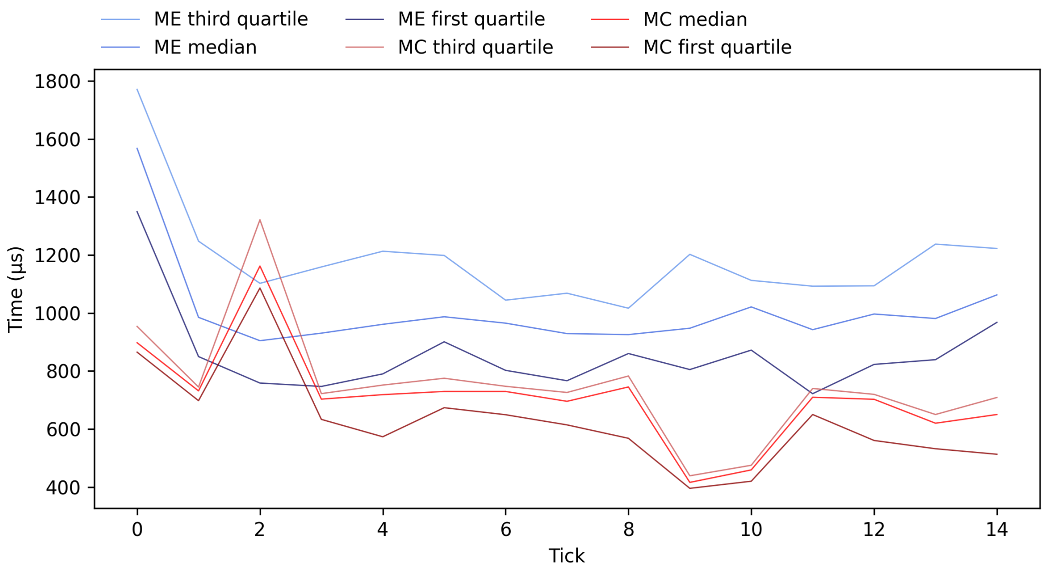


(b) Histogram of aggregated distribution

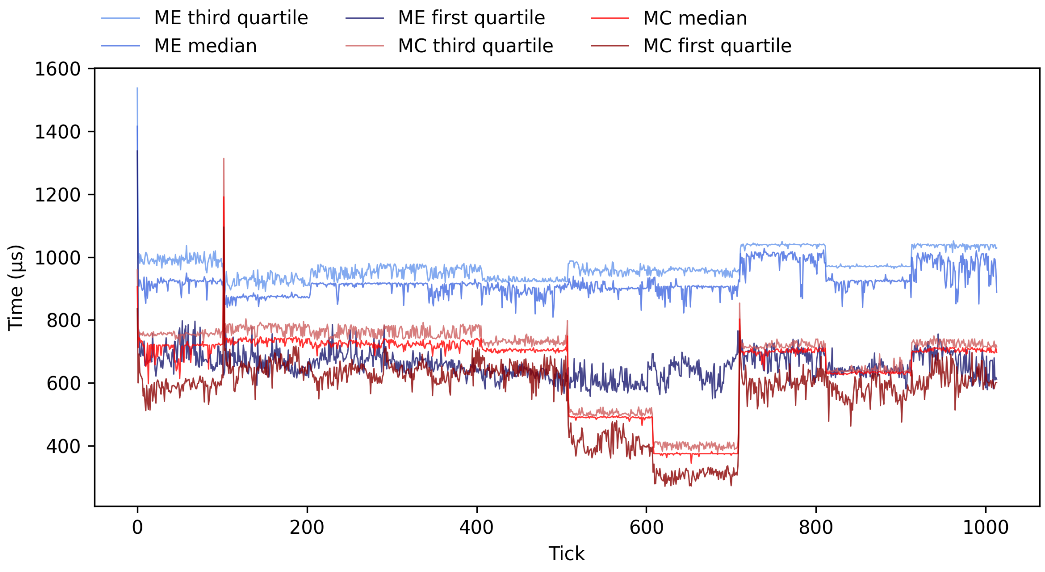
Figure 4.10. Tick times of the Barcode model

300 microseconds in more than half (but less than a three quarters) of all ticks. The latter hypothesis may also explain why MCS is less affected by this phenomenon: if interruptions occur somewhat periodically, then a faster tick function is less likely to be interrupted. Naturally, the two explanations are not mutually exclusive; they may both contribute to the phenomenon.

Furthermore, Figure 4.11b shows that even MES's tick time is influenced by internal state: for



(a) Quartiles for the dense trace



(b) Quartiles for the sparse trace

Figure 4.11. Tick times of the Backhoe model

instance, median tick time is significantly higher between ticks 700 and 800 than in the subsequent 100 ticks. This is an at least somewhat surprising result for the netlist-based synthesis because of its usually high tick-by-tick execution time stability discussed in Section 2.1.1.

Figure 4.12 illustrates that MCS and MES produce a similar distribution of tick times with clearly distinguishable clusters in the low end and a high concentration around the median. The clusters produced by MCS are more centered, though, with visible spikes in the histogram.

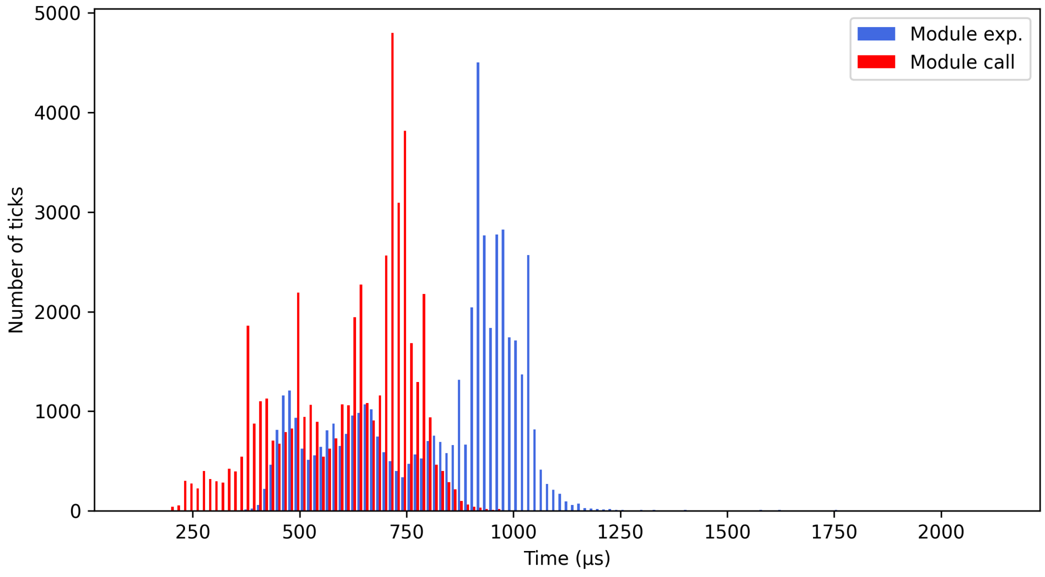


Figure 4.12. Histogram of Backhoe sparse trace aggregated tick time distribution

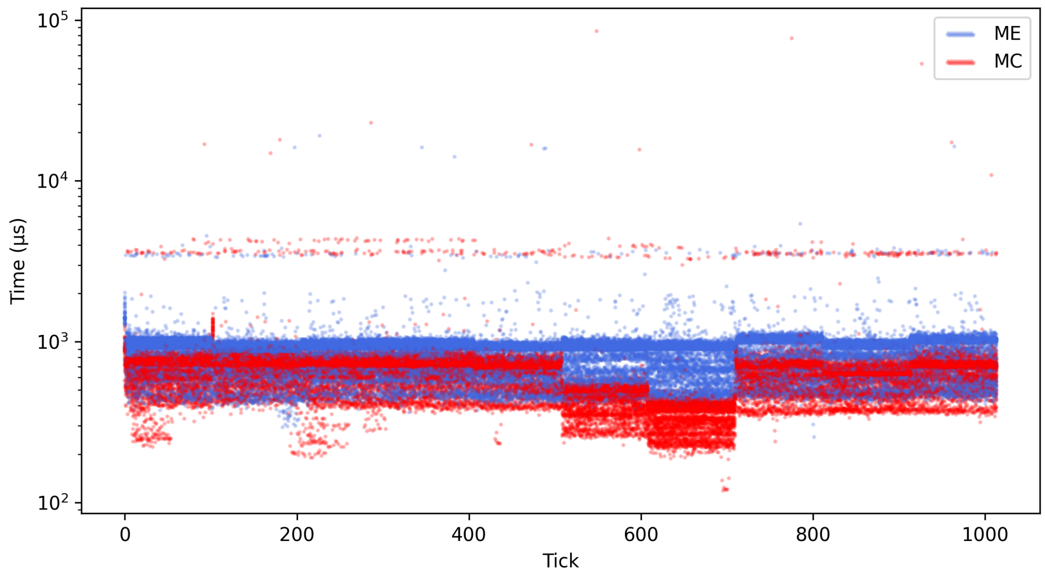


Figure 4.13. Scatterplot of all Backhoe sparse trace tick times *before* outlier removal (log scale)

A look at the scatterplot of all tick times (before outlier removal; Figure 4.13) reveals that these clusters are not concentrated in time, as one might suspect, but rather spread out over time in *layers* hundreds of ticks in width.

Furthermore, there are two distinguishable bands of tick times between 2.5 and 3.5 milliseconds, most of which, but not all, from executions of MCS. These fall just outside of the $3 \cdot$ IQR radius that is used to detect outliers, which is why they do not appear in the histogram nor do they influence the calculated mean tick time. While fewer MES ticks fall into these exact bands, many occur between 1 and 2 milliseconds, a zone almost free of MCS ticks. This leads to the conjecture that *if* an MCS tick is interrupted or takes, for any reason, at least a millisecond to compute, it is much more likely to be interrupted for long enough to take at least 2.5 milliseconds.

Finally, after the first (resp. 200th) tick, MCS produces a curious amount of as of yet inexplicable lower outliers.

Counter. Figure 4.14 shows the tick times for the Counter model. It is immediately apparent that the tick times of MCS are consistently, almost strictly, smaller than those of MES. Additionally, MCS does not display the same band of outliers as it does in the Backhoe model (cf. Figure 4.13).

Median times of the MES experiments oscillate around the 5.5 millisecond mark with an amplitude of approximately 500 microseconds and a period of exactly 8 ticks, as can be seen in Figure 4.14a. The period is remarkable because *once every 8 ticks* is also the frequency of the output of the divided clock module (cf. Figure 4.6e); the output is false for four ticks, then true for the next four. In other words: the binary counter module *counts* every 8th tick. The exact cause of this oscillation is not known, however. Remarkably, MCS seems less affected by the phenomenon; while some periodicity is undeniably observable in its tick times, it is often not significant enough to be visible above the noise of random tick time fluctuations.

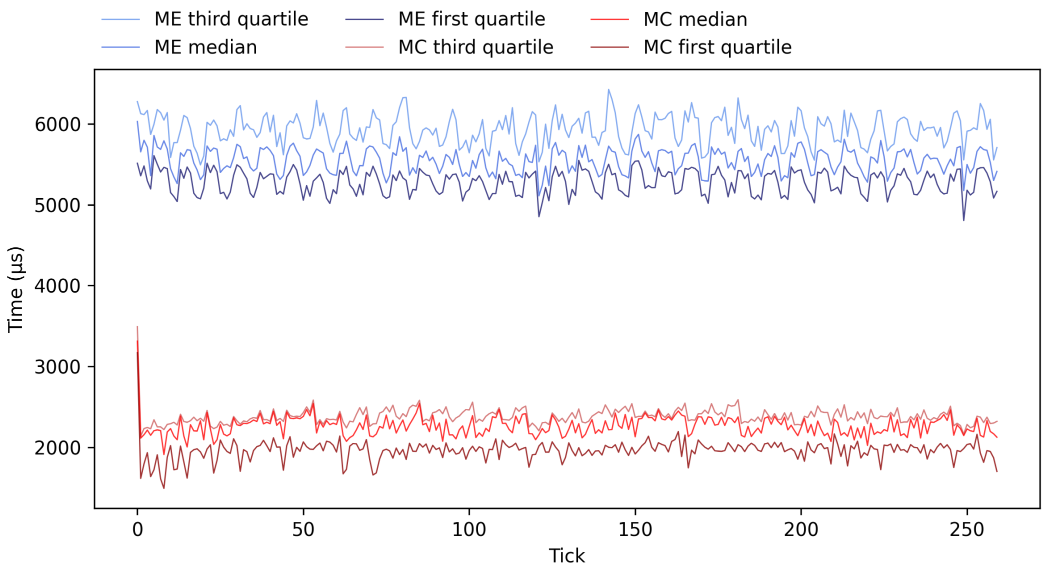
Furthermore, MES displays two clearly visible tick time minima: the first one in tick 121, the second one in tick 249. The trace reveals that these are the ticks in which the first four (resp. five) registers overflow, going from 1 to 0. Clearly, and contrary to intuition, the counter overflow causes execution time to experience a downward spike. The effect is weak, though, and it cannot be said with certainty whether or not MCS is affected as well.

The scatterplot in Figure 4.14b shows that the Counter model displays the same phenomenon of lower outliers as the Backhoe model, though it seems to affect MCS with a larger magnitude; while outliers have a similar (logarithmic) distance from the median as those produced by MES, there are visibly more of them and they form an almost continuous band.

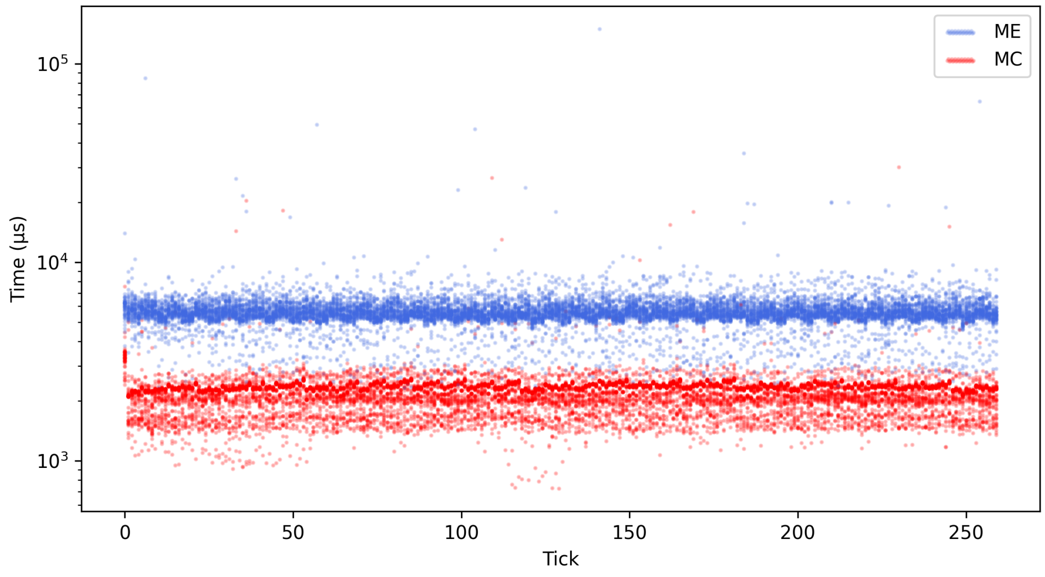
Counter variations. The two variations on the Counter model (Figure 4.15) exhibit the same phenomena as the original, albeit adjusted to the models'individual characteristics. The period of the Counter (full frequency) model's low tick time spikes, for instance, is 64, although less distinguished spikes often appear exactly at the half way point between two more visible ones. As this model's internal clock *ticks* with a frequency four times that of the original, the natural assumption is that its minima should occur every 32nd tick, yet this assumption is only partially true. Additionally, the model's clock period of two ticks does not provide the resolution necessary to visually confirm whether or not the low-level oscillation observed in the original model exists here.

The Counter (half width) model displays more erratic tick time behavior along the tick axis, although MES's per-tick variance is visibly smaller across runs when compared to the full-size model, even when measured in relation to the mean. MCS does not share this phenomenon. Overall, MES's tick times increase approximately linearly with model size between the smaller and the two larger models, while MCS's increase only slightly.

Code and executable size. Arguably more important for embedded systems than performance is, from an economic perspective, the issue of size, specifically the size of the compiled executable. Because embedded systems use a large variety of different microcontroller and CPU architectures and compilers, the size of a binary executable compiled for any one system is not necessarily representative



(a) Tick time quartiles

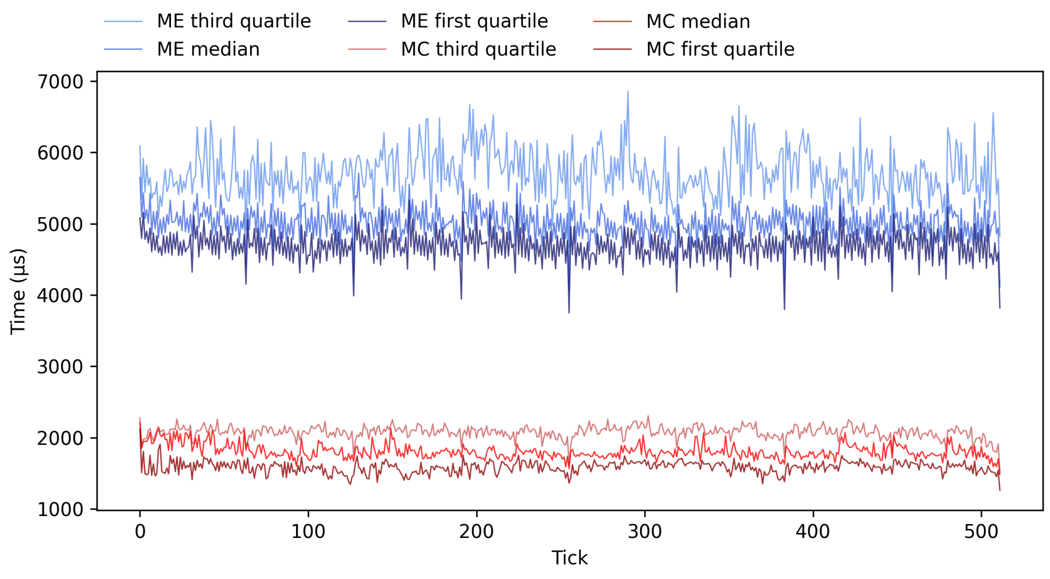


(b) Scatterplot of all tick times

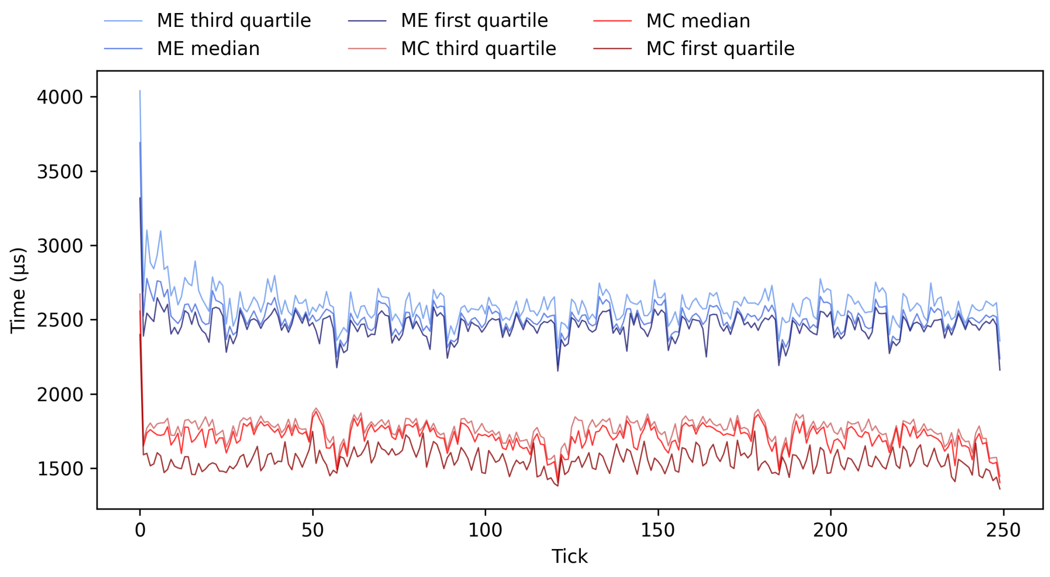
Figure 4.14. Tick times of the Counter model

for all systems.

It is therefore advisable to consider a size metric common to all systems: the number of lines of the generated C code, as shown in Figure 4.16. The numbers displayed in the figure are determined by counting all lines of code that are neither empty nor comments. While the size overhead introduced by its proxy states puts MCS at a clear disadvantage for the two smaller models, it seems that MCS's



(a) Tick times of the Counter (full freq.) model



(b) Tick times of the Counter (half width) model

Figure 4.15. Tick time quartiles of the Counter variations

generated code size grows more slowly than MES's does. Even the relatively small Counter (half width) model is significantly smaller when using MCS. This is no doubt because of its heavy re-use of references. The most striking difference between the two approaches'size implications is visible when comparing the above-mentioned model with Counter (full frequency): instead of nearly doubling in size, as the version using MES does, it decreases in size. Reminding oneself of the structure of the larger

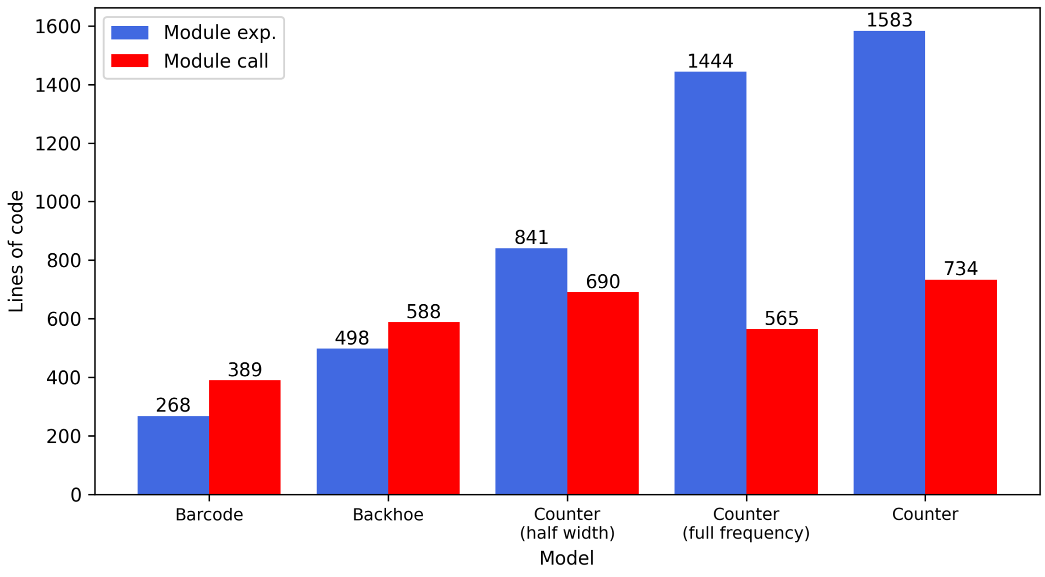


Figure 4.16. Number of lines of generated code, excl. empty lines or comments, by model and approach

model, the reason for this disparity is clear; the larger model does not contain any more unique references than the smaller one, in fact it has one less. The overhead introduced by the additional non-unique references, it follows, is relatively insignificant.

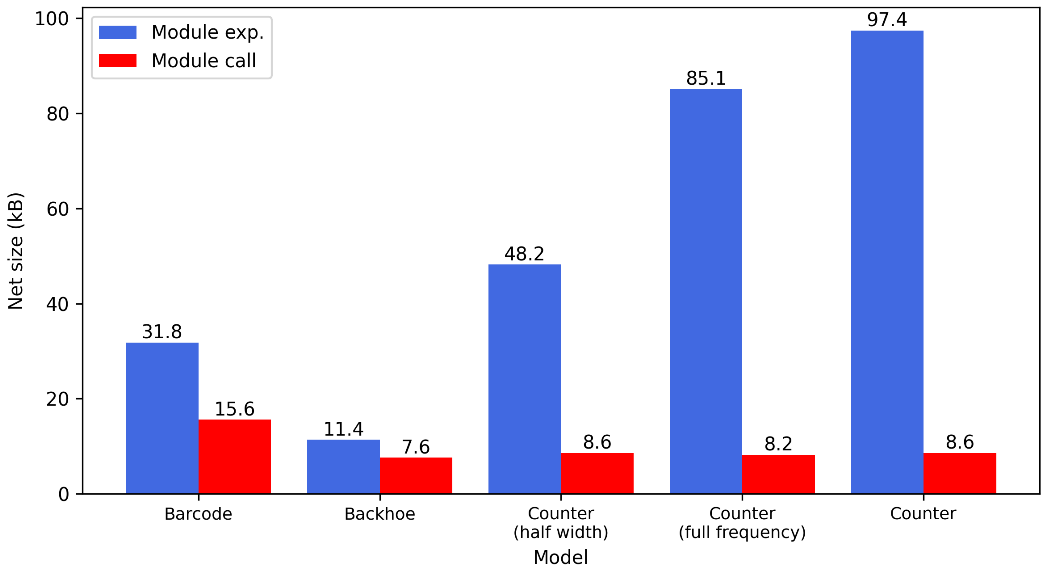


Figure 4.17. Net compiled executable size by model and approach

As a single line of code can, however, amount to any number of processor instructions after compilation, it would be careless not to examine exemplary binary executables. SCCharts do not generally run as a standalone application; they are instead called from within a host environment. Therefore, it is difficult to obtain accurate size information from generated code alone. The solution chosen here is to evaluate the size of the executables used in the performance benchmark as detailed above. Because a significant number of instructions are used to gather internal model information, an *empty* SCCharts model was first compiled for simulation and its size, 46.6 kilobytes, subtracted from all measured executable sizes in order to arrive at the *net* size, i. e., the size of only the model, shown in Figure 4.17. As the netlist synthesis does not rely on external libraries, the size obtained this way can be regarded as an accurate approximation of the actual compiled model size. However, this does not imply that the results presented here hold for other architectures, compilers, or even optimization flags.

The space saving potential of MCS is very promising regardless: the resulting executable is smaller for every tested model. Especially the `Counter` models demonstrate MCS's advantage for models featuring heavy code re-use.

Alternative compiler configurations. As mentioned above, model performance relies heavily on compile time optimization. While the optimization level `-O3` has been chosen in order to achieve best-case performance for both reference resolution approaches, different configurations may be used in practice. In order to reflect this circumstance, an alternative benchmark using `-O0` optimization was conducted, the results of which are summarized in Figures 4.18-4.22 below.

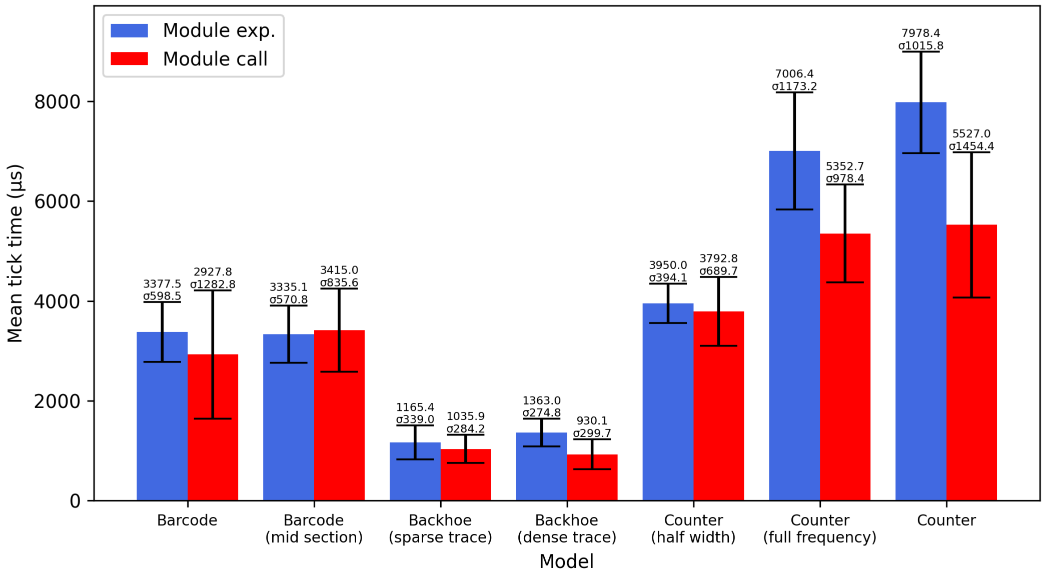


Figure 4.18. Mean aggregated tick times and tick time standard deviation for all traces when compiled with `-O0`

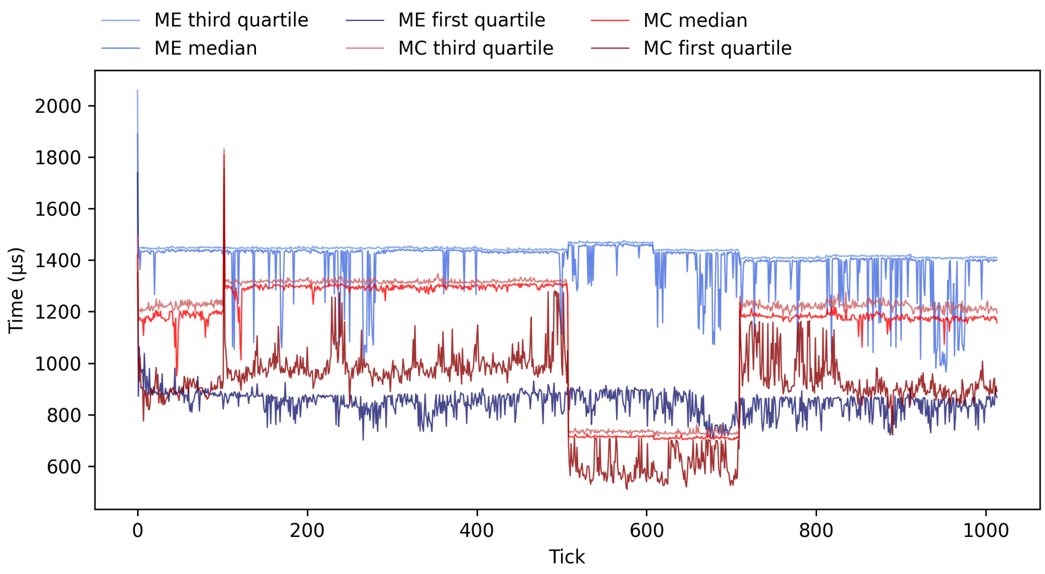


Figure 4.19. Tick time quartiles for the `Backhoe` model (sparse trace) when compiled with `-O0`

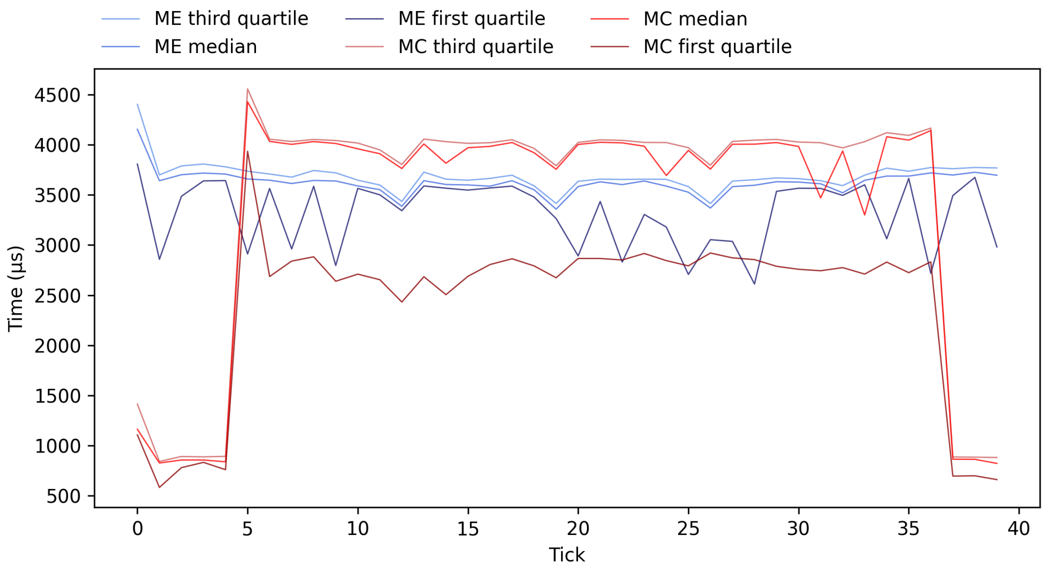


Figure 4.20. Tick time quartiles for the `Barcode` model when compiled with `-O0`

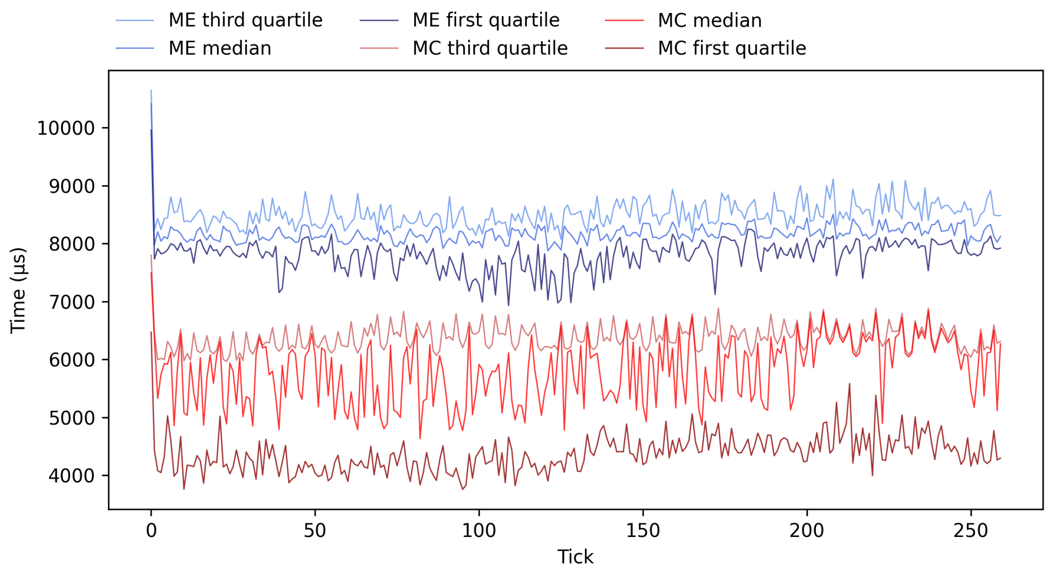


Figure 4.21. Tick time quartiles for the `Counter` model when compiled with `-O0`

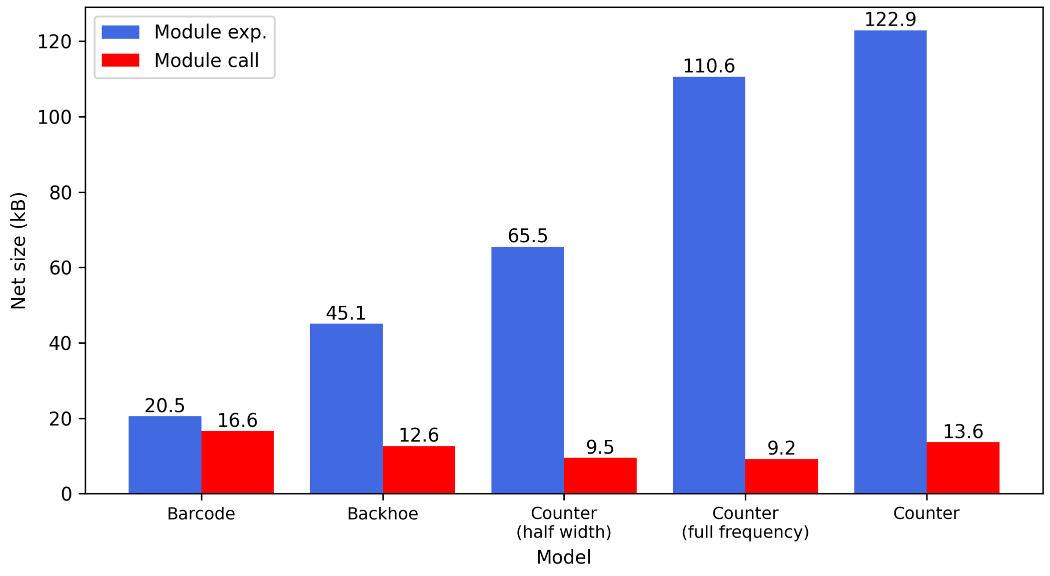


Figure 4.22. Net executable sizes for all models when compiled with `-O0`

4.4 Discussion

The comparative benchmark, though limited in scope, has shown that MCS has a significant performance advantage over MES for models that heavily re-use references. Surprisingly, even for models that do not re-use references at all, MCS produces much smaller executables. Though both approaches exhibit somewhat unpredictable execution time behavior in various situations, there is good indication that MCS is at least as reliable w.r.t. execution time as MES. Due to its call-by-value design, MCS has the disadvantage that back-and-forth communication between caller and callee cause the model to be rejected as discussed in Section 4.2. Solving this problem requires a more sophisticated caller-callee interface. Overall, using MCS—provided the scheduling constraints allow for it—is recommended if a model:

(i) references the same model in multiple places;

(ii) references a complex model that is inactive in a significant proportion of ticks;

(iii) is required to use pre-compiled models; or

(iv) otherwise produces too large executables.

Recall that MCS and MES are not mutually exclusive. Conceptually, the same model can use both types of reference. Therefore, the choice between MCS and MES can be made on the reference level. Nevertheless, MCS could become the default choice.

Chapter 5

Conclusion

In this chapter, the thesis is summarized and further work is pointed out. Section 5.1 recapitulates the results of this theses. Section 5.2 discusses possible avenues for future research.

5.1 Summary

MCS is a promising alternative to MES. It enables modular code generation for SCCharts using black-box scheduling. Modules can be separately compiled and later included in any model. This is accomplished by generating a separate tick and reset function for each module. The caller is then transformed by introducing a proxy class for each unique callee module and a proxy state, which communicates control-flow between caller and callee, for each instance.

An implementation into KIELER has been developed for this thesis. It allows generation of C code using the netlist-based synthesis. The transformation uses a two-pronged approach, introducing the proxy class and states in the first transformation and adapting the syntax to the target language in the second. Most data types and SCCharts model elements are supported. Although MCS is syntactically a drop-in replacement for MES, not all models that can be compiled using MES can be compiled with MCS. In particular, instantaneous back-and-forth communication between modules is prohibited by the black-box approach.

Nevertheless, MCS has proven to be advantageous for run-time performance and executable size. All but the very smallest models benefit from significantly improved median execution times. Furthermore, every tested MCS model produces a smaller executable than its respective MES counterpart. These advantages make MCS a good choice as the default approach for referenced SCCharts.

5.2 Future Work

Sections 4.1 and 4.2 of this thesis have already pointed out a number of open questions, as well as directions in which MCS could develop. This section expands upon these questions and suggestions and gives solution approaches where possible.

5.2.1 Instantaneous Bidirectional Communication

As discussed in Section 4.2, MCS precludes instantaneous bidirectional communication. Simply stated, a callee cannot ask for input and receive an answer within the same tick. It would be worthwhile to investigate how essential design patterns that requires instantaneous bidirectional communication are for large real-world SC models.

This could be accomplished by re-designing the existing model railway controller [SMS+15]—or even building a new one from scratch—using only MCS wherever possible. Doing so would also certainly provide a model for benchmarking MCS that is orders of magnitude larger than all models tested thus far.

5.2.2 Array Passing

Arrays and other arbitrary-size data structures present a problem for MCS. Passing arrays to and from callees by copying all values would be consistent with the existing call-by-value design philosophy. However, doing so could waste an inordinate amount of processor cycles copying unchanged—and unneeded—array values. Passing arrays by reference is also problematic. This approach would require explicit scheduling directives in order to ensure that all array values are written before a callee's tick function is called—and that array values are only read after the callee has finished its work.

Arguably, both approaches are insufficient. The call-by-value approach, however, is much more consistent with the rest of MCS. It is the author's opinion that it should be applied to arrays as well, mitigating performance penalties wherever possible. Perhaps it is feasible to statically determine array accesses and copy only values that may have changed. Such scheduling information could be distributed alongside pre-compiled modules in their respective header files. A more primitive solution would be the transformation of arrays into separate variables. Unused array fields could then be eliminated from a module's interface.

5.2.3 Synthesis and Target Language Support

The current implementation, which is described in Chapter 3, only supports the netlist-based synthesis and C code generation. This is not a conceptual limitation of MCS. On the contrary: MCS is compatible with a range of other synthesis approaches and target languages.

The generation of Java code from models using MCS could be accomplished by implementing a variant of the `ReferenceCallProcessor` that transforms SCG nodes such that Java syntax is used instead of C syntax. The priority-based synthesis could be made to support MCS by modifying its SCG transformations and code generation processor such that multiple SCGs are processed independently from one another. The lean state-based synthesis, which does not use SCGs as an intermediate format, would require the methods introduced in the MCS SCCharts transformation to be implemented. This could be accomplished within the scope of the library functions that the lean state-based synthesis uses.

5.2.4 Gray-box Scheduling

While MCS is conceptually a black-box approach, it uses techniques that could be considered gray-boxing. For instance, the optional **`@isDelayed`** and **`@nonFinal`** annotations give the compiler information about the inner behavior of a callee. Introducing further gray-box techniques could allow for some interleaving of control flow between caller and callee. This could enable instantaneous bidirectional communication in a limited capacity.

Bibliography

[And03] Charles André. *Semantics of SyncCharts*. Tech. rep. ISRN I3S/RR–2003–24–FR. Sophia-Antipolis, France: I3S Laboratory, Apr. 2003.

[Ber00] Gérard Berry. “The Foundations of Esterel”. In: *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. Cambridge, MA, USA: MIT Press, 2000, pp. 425–454. ISBN: 0-262-16188-5.

[CPH+87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. “LUSTRE: a declarative language for programming synchronous systems”. In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL’87)*. Munich, Germany: ACM, 1987, pp. 178–188.

[Est08] Esterel-Technologies. In: *The Esterel v7 Reference Manual Version v7.60* (Nov. 2008).

[GG18] Friedrich Gretz and Franz-Josef Grosch. “Blech, Imperative Synchronous Programming!” In: *Proc. Forum on Specification Design Languages (FDL’ 18)*. Sept. 2018, pp. 5–16. DOI: 10.1109/FDL.2018.8524036.

[GGM+20] F. Gretz, F-J. Grosch, M. Mendler, and S. Scheele. “Synchronized Shared Memory and Procedural Abstraction: Towards a Formal Semantics of Blech”. In: *2020 Forum for Specification and Design Languages (FDL)*. 2020, pp. 1–8. DOI: 10.1109/FDL50818.2020.9232942.

[Har87] David Harel. “Statecharts: A Visual Formalism for Complex Systems”. In: *Science of Computer Programming* 8.3 (June 1987), pp. 231–274.

[HDM+13] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O’Brien. *SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications*. Technical Report 1311. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2013.

[HMA+14] Reinhard Von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O’brien, and Partha Roop. “Sequentially Constructive Concurrency—A Conservative Extension of the Synchronous Model of Computation”. In: *ACM Trans. Embed. Comput. Syst.* 13.4s (July 2014). ISSN: 1539-9087. DOI: 10.1145/2627350. URL: https://doi.org/10.1145/2627350.

[HPL+99] Olivier Hainque, Laurent Pautet, Yann Le Biannic, and Éric Nassor. “Cronos: a separate compilation tool set for modular esterel applications”. In: *FM’99 — Formal Methods*. Ed. by Jeannette M. Wing, Jim Woodcock, and Jim Davies. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 1836–1853. ISBN: 978-3-540-48118-8.

[LST09] Roberto Lublinerman, Christian Szegedy, and Stavros Tripakis. “Modular Code Generation from Synchronous Block Diagrams: Modularity vs. Code Size”. In: POPL ’09. Savannah, GA, USA: Association for Computing Machinery, 2009, pp. 78–89. ISBN: 9781605583792. DOI: 10.1145/1480881.1480893. URL: https://doi.org/10.1145/1480881.1480893.

[Mot17] Christian Motika. *SCCharts – Language and Interactive Incremental Compilation*. Kiel Computer Science Series 2017/2. Dissertation, Faculty of Engineering, Kiel University, Germany. Department of Computer Science, 2017. ISBN: 9783746009391. DOI: 10.21941/kcss/2017/02.

[Pei17] Lars Peiler. “Priority-based Compilation of SCCharts”. `https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lpe-mt.pdf`. Master thesis. Kiel University, Department of Computer Science, Oct. 2017.

[RH18] Peter J. Rousseeuw and Mia Hubert. “Anomaly detection by robust statistics”. In: *WIREs Data Mining and Knowledge Discovery* 8.2 (2018), e1236. DOI: `https://doi.org/10.1002/widm.1236`. eprint: `https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/widm.1236`. URL: `https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/widm.1236`.

[SMS+15] Steven Smyth, Christian Motika, Alexander Schulz-Rosengarten, Nis Boerge Wechselberg, Carsten Sprung, and Reinhard von Hanxleden. *SCCharts: The Railway Project Report*. Technical Report 1510. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Aug. 2015.

[SMS+19] Steven Smyth, Christian Motika, Alexander Schulz-Rosengarten, Sören Domrös, Lena Grimm, Andreas Stange, and Reinhard von Hanxleden. *SCCharts: The Mindstorms Report*. Technical Report 1904. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2019.

[Smy21] Steven Smyth. *Interactive Model -Based Compilation — A Modeller -Driven Development Approach*. Kiel Computer Science Series 2021/1. Dissertation , Faculty of Engineering , Kiel University. Department of Computer Science , CAU Kiel, 2021. DOI: `10.21941/kcss/2021/1`.

[SSH18] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. “Towards Interactive Compilation Models”. In: *Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018)*. Vol. 11244. LNCS. Limassol, Cyprus: Springer, Nov. 2018, pp. 246–260.

[SW87] Richard M Stallman and Zachary Weinberg. “The C preprocessor”. In: *Free Software Foundation* (1987), p. 8.