Towards Object-Oriented Modeling in SCCharts

Alexander Schulz-Rosengarten, Steven Smyth
Department of Computer Science
Kiel University
Kiel, Germany
{als, ssm}@informatik.uni-kiel.de

Michael Mendler
Faculty of Information Systems and Applied Computer Sciences
Bamberg University
Bamberg, Germany
michael.mendler@uni-bamberg.de

***Abstract*—Object orientation is a powerful and widely used paradigm for abstraction and structuring in programming. Many languages are designed with this principle or support different degrees of object orientation. In synchronous languages, originally developed to design embedded reactive systems, there are only few object-oriented influences. However, especially in combination with a statechart notation, the modeling process can be improved by facilitating object orientation as we argue here. At the same time the graphical representation can be used to illustrate the object-oriented design of a system.**

Synchronous statechart dialects, such as the SCCharts language, provide deterministic concurrency for specifying safety-critical systems. Using SCCharts as an example, we illustrate how an object-oriented modeling approach that supports inheritance, can be introduced. We further present how external, i. e. host language, objects can be included in the SCCharts language. Specifically, we discuss how the recently developed concepts of *scheduling directives* and *scheduling policies* can be used to ensure the determinism of objects while retaining encapsulation.

***Index Terms*—Synchronous languages, object orientation, inheritance, determinacy, state machine modeling**

I. Introduction

The object-oriented (OO) paradigm has proven to be a powerful design and programming concept that facilitates an abstract and modular design of large and complex systems. Consequently, most general-purpose programming languages popular today support OO concepts, such as encapsulation of data and functions, inheritance on abstract data types and message passing. In software engineering, the OO paradigm is often combined with a model-based approach, for example in UML, to create well-designed software architectures. Today, software engineers are well-trained in Java and C++ programming, so that OO design techniques are second nature to them. Hence, it is compelling to try and exploit the benefits of OO also in a specialized domain such as synchronous programming.

Synchronous languages (SLs) are designed for the programming of safety-critical embedded systems. These typically involve complex interactions between system components and the environment, while imposing stringent requirements on functional correctness, real-time performance and fault tolerance. SLs are especially suited for that task, as they provide deterministic and simple mathematical semantics based on Mealy machines. One of the key issues addressed by SLs is the safe handling of concurrency, which is both a powerful programming principle and intrinsic to the execution model for reactive embedded systems. Concurrency can easily compromise a safety-critical system by introducing *race conditions*. SLs solve this problem traditionally by two techniques. Firstly, concurrent threads are forced to operate in lock-step, by synchronizing on a logical clock. The clock acts as a global barrier that breaks the computation into a sequence of reaction *instants*. Secondly, during each instant, concurrent threads may only communicate through synchronous *signals* or *channels*. These special-purpose shared memory structures are protected by an *(intra-instant) synchronization protocol* which ensures a unique value per instant, despite possibly multiple concurrent write and read accesses. As a result, the observable behavior of a program is that of a synchronous Mealy machine, providing a deterministic functional reaction to its environment. The compiler performs a static *causality analysis* in order to establish that a program is schedulable under the synchronization protocol. If a scheduling order can be found, it is considered *constructive* otherwise rejected.

SLs come in different programming styles and with slightly different synchronization protocols. The most prominent SLs are Esterel [5] and Lustre [14]. Lustre is based on data-flow equations and is commercially used by the Safety-Critical Application Development Environment (SCADE) [10] that allows the graphical modeling of data-flow diagrams. In Lustre and its derivatives communication is via single-writer/multi-reader (1-place) channels. Esterel supports an imperative coding style and uses multi-writer/multi-reader signals, implementing the *write-before-read* protocol, which aggregates all written values before reading. This protocol prohibits a shared signal to be overwritten during an instant, only *thread-local variable* can be destructively updated but cannot be shared concurrently. SyncCharts [3] is a dialect of Harel's statecharts with the synchronous semantics of Esterel. Sequentially Constructive Statecharts (SCCharts) [23] provide a statecharts notation inspired by SyncCharts but relax the *write-before-read* protocol so that signals, channels and local variables are unified in a single notion, the *SC-variable*. SC-variables are synchronized under the initialize-update-read (iur) protocol which supports at the same time concurrent multi-writer/multi-reader accesses and destructive updates by a single thread. This respects the sequential ordering of statements while still preserving deterministic concurrency.

Traditionally, SLs are used with rather low-level target platforms, such as micro-controllers often programmed in subsets of C. In such contexts there is no strong need for

978-1-7281-4113-8/19/$31.00

OO design concepts. However, SLs are also used as high-level orchestration languages to control a larger software system deterministically. This requires a close and convenient integration with the targeted host languages, such as Java or C/C++. In the safety-critical domain such systems must satisfy high standards regarding system architecture, documentation and code reviews. As experienced in other programming domains shows, OO has a lot to offer here. Also, object-based modelling has already long been recognised as a useful structuring principle in intermediate languages for the modular compilation of traditional SLs [13], [6]. Despite this, however, the OO paradigm has not yet been made available at the source level for the programmer in leading SLs.

Without entering into a wide-ranging discussion about OO programming, we aim to enrich the embedded safety-critical domain of SLs by OO facilities, as far as they fit. In this paper, we are adding to SCCharts selected OO features that are not only useful but also can be compiled conservatively by simple semantic transformations that can be inspected and verified on source level, thus grounding their semantics in the existing well-established execution model of SCCharts. In this way, we provide some of the benefits of OO modeling yet remain on safe semantical terrain.

Contributions and Outline

We first discuss related work in Sec. II and then present the following contributions:

- We present how the OO design is applied to the modeling concepts of SCCharts (Sec. III). SCCharts are extended by inheritance to facilitate abstraction and reusability, as well as the possibility to design classes.
- We show how class-based data structures coming from a host language can be used in SCCharts (Sec. IV).
- We propose mechanisms to ensure the determinism of objects (Sec. V) while retaining their encapsulation under a "black box" scheduling approach. Specifically, we present solutions based on *scheduling directives* (Sec. V-A) and *scheduling policies* (Sec. V-B).

We conclude and give a short outlook in Sec. VI.

II. Related Work

There is much work on adaptations of OO concepts into various statechart dialects and SLs. André et al. [2] introduce *synchronous objects* (referred here as SOs) based on the idea of the *reactive object model* [7]. This approach divides the program into a collection of *regular* host code objects (referred here as ROs) and SOs that communicate with each other. Messaging allows SOs to communicate instantaneously and preserves the synchronous semantics. The resulting directed interconnection graph is required to be acyclic because modules are considered "black boxes" which cannot interleave with each other. Communication with ROs is done via signals that can be read outside SOs but inputs require special handling by *interface objects* to enter the synchronous messaging mechanism. The structure of a system using SOs is represented by an object-modeling technique (OMT) class diagram augmented by communication interfaces. For specifying the internal behavior of an SO, André et al. support SyncCharts, among other SLs. SCCharts could be integrated, too, by providing a code generation that synthesizes SOs from SCCharts modules. In contrast to the approach presented here, however, the SyncCharts dialect is not extended by OO features but the models are synthesized into separate interconnected objects in an OO target language (C++). Also, André et al. do not address the integration of more complex OO data structures and the preservation of determinism in instantaneous communication, as we do here.

The synchronous OO language *synERJY* [8] provides synchronous reactive classes in a Java-like syntax. Programs can be written as imperative code, dataflow equations or textual state machines. The resulting synchronous reactive objects communicate only via signals with each other or the environment, similar to SOs by André et al. For handling causality problems inside a synchronous reactive class, synERJY provides a mechanism for specifying static precedences, that is very similar to Scheduling Directives (SDs). However, it does not include a combination with graphical modeling or ways to determinstically include host code objects.

The SL Blech [12], which has recently been introduced in an industrial context, provides shared data structures of a general form with a deterministic reference mechanism and modular ("black box") scheduling. It uses a rigid form of scheduling which is not as flexible as our scheduling directives or as expressive as our policies. Also, Blech does not support OO concepts such as inheritance.

Furthermore, there are many concepts for OO modeling using statecharts, without an explicitly synchronous semantics. *ObjectCharts* [11] are developed by Coleman et al. and characterize the communication behavior of an object as a state machine based on Harel's statechart diagrams. In combination with a *configuration diagram* that specifies the object relations such as instancing, inheritance, and communication, they allow a top-down design of a system in an iterative development process. A similar approach is presented by Harel and Gery with *O-charts* for specifying classes and structures and statecharts for modeling the behavior of objects [15]. They introduce an OO statecharts version, which resulted in the *Rhapsody* semantics of statecharts [16] and an adoption by UML. These statecharts support C++ code for specifying transition actions and provide inheritance that allows refining inherited statecharts by decomposing states or adding orthogonal states, as well as adding new transitions or modifying existing ones. *ROOMcharts* [21] is another statecharts dialect, used in the real-time object-oriented modeling (ROOM) language, to specify the behavior of actors in the higher-level ROOM model. ROOMcharts also support inheritance but prohibit concurrency since the authors consider synchronization mechanisms too inefficient for the targeted real-time domain. With SCCharts being a statecharts dialect, the mentioned dialects and their introduction of OO modeling inspired the concepts presented here, disregarding the differences in semantics. Furthermore, we do not follow the approach of separating the modeling

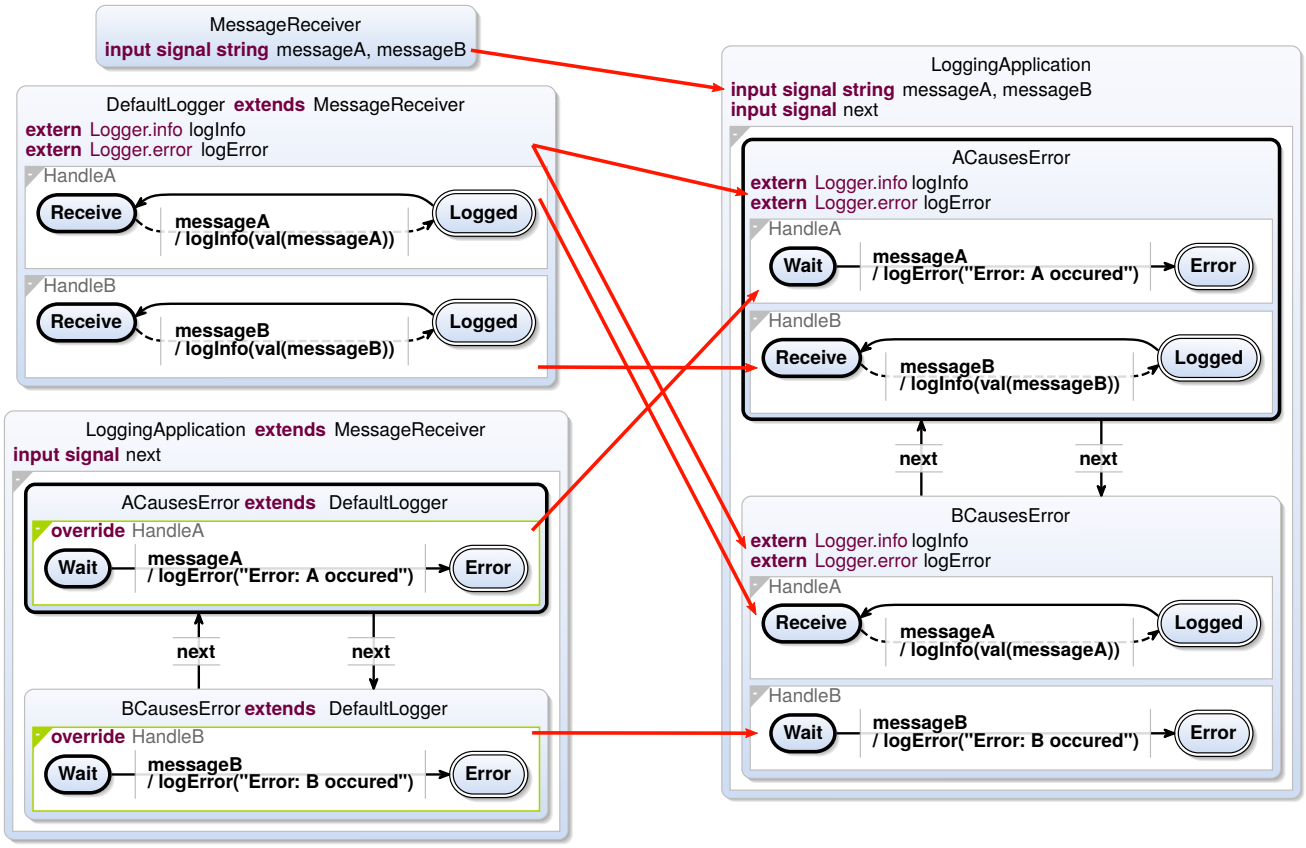


Fig. 1: Example for usage of inheritance in SCCharts (left) and the result after inheritance is statically expanded by the compiler (right). Red arrows indicate where the parts of the model are expanded into.

of the system's structure from the modeling of behavior, as discussed in Sec. III-C.

Lohstroh et. al [17] propose a real time refinement of the actor model, called *reactors*. A reactor is a computation unit that manages its own state and exposes multiple reactions like an object exposes its methods. Each reaction is associated with a time-stamped input, clock or a trigger action. Reactions are sequentially ordered and executed atomically to ensure determinism. A reaction is considered logically instantaneous at the time stamp of its trigger. This generates a notion of "logical time," like a synchronous instant in SCCharts. Reactors do not communicate by method calls, such as we propose here, but via inputs and outputs events to exchange messages similar to Esterel signals or Lustre data flow channels.

III. Object-Oriented Modeling in SCCharts

The OO paradigm includes many aspects that can greatly improve the design, quality, and understandability of programs, such as object composition, encapsulation, and inheritance. Not surprisingly, including OO features can greatly improve the effectiveness and convenience when modeling in SCCharts.

A. Inheritance

Inheritance is an OO core concept to express commonalities between entities of a system. The need for such a feature was recently expressed by students using SCCharts in an educational project for modeling a larger scale controller for a model railway installation[1]. SCCharts provide a macro expansion mechanism, similar to Esterel [4] and SyncCharts, which allows to include code from another SCChart. However, this requires to redefine and bind interfaces in every module again, complicating the design of commonalities. For example, it also hampers the modeling of different types of trains that adapt the same basic behavior in different ways. This motivates us to extend SCCharts further towards OO and to introduce *inheritance*.

Inheritance in SCCharts now allows to derive states from one or more *base states*[2] using the extend keyword. A state inherits all variables and behavior from all its base states. In principle, inheritance in SCCharts is an advanced macro expansion that statically expands base states. Inheritance unfolds its full potential when allowing overriding the inherited behavior to adapt it for the purpose of the extending object. Harel's statecharts support fine granular altering of states and transitions in extending statecharts. However, here we follow a more conservative approach and allow only *region overriding*.

[1] https://rtsys.informatik.uni-kiel.de/confluence/x/VABgAQ

[2] The term 'superstate', following superclass, might be more obvious here. However, in SCCharts a superstate is already defined as a state that contains inner behavior such as regions [23].

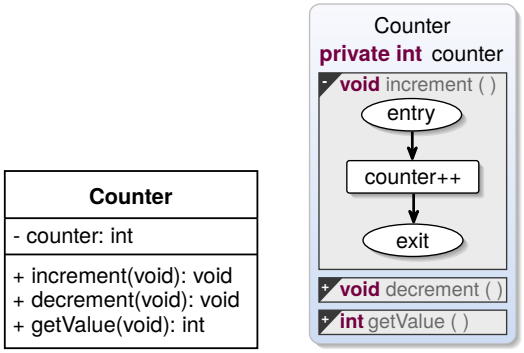
Fig. 1 shows on the left-hand side how inheritance can be used in SCCharts. The SCChart is a simplified model of a real-world example. In the underlying scenario, incoming messages, here messageA and messageB, must be processed differently depending on the state of the application. By default, a receive message must be logged. This common behavior is modeled in the DefaultLogger, which has separate regions for each message. The state machine in these regions immediately logs the message content on an info level if received, switches to the Logged state, and returns to the receiving state in the next instant to process further messages. The input messages are declared in MessageReceiver and available due to inheritance. In the actual application represented by LoggingApplication, the behavior differs from the default logging behavior depending on the state. In this abstract example there are two states in the application, ACausesError and BCausesError, that alternate triggered by the next input. Each state inherits the behavior of the DefaultLogger. In state ACausesError the handling of messageA is altered by overriding region HandleA, indicated by the override keyword and the green color. If messageA is received, an error is logged and the Error state is entered but not left until next occurs, ignoring future occurrences of messageA. Analogously the state BCausesError is designed for MessageB.

Inheritance is considered an extended feature [23] in SCCharts and removed by a model-to-model transformation in the first step of the compilation. Fig. 1 presents on the right-hand side the result of this transformation, which is essentially a macro expansion with finer granularity, all variables and regions are copied into their extending states w. r. t. overriding. The red arrows indicate this process. In a macro expansion, input and output variables must be bound, since only top level SCCharts can have an input output interface. Inheritance handles this aspect in the same way. In this example the input messages declared in MessageReceiver and inherited by DefaultLogger are automatically bound to the input messages of LoggingApplication since they share the same base state.

In general, if any conflicts or inconsistencies in the inheritance hierarchy or name clashes occur the compiler rejects the model, similar to default methods in Java interfaces. Furthermore, we added access modifiers for variables and regions to allow visibility restrictions and support accessing the scope of the base state while overriding, by using the super keyword, as known from languages such as Java. We contributed the implementation of inheritance to the open-source KIELER tool providing SCCharts.

B. Class Modeling

Traditionally, SCCharts are primarily used to model a system as a statechart. However, in line with the new OO view that we wish to advance here, SCCharts can also be used to model complex data structures. In SCCharts a state has a name, can contain variables and provides behavior usually associated with its regions. However, the behavior in regions is always executed when the state is active. To adhere to the concept of objects we introduce *methods* in SCCharts that are only executed when invoked and can be modeled alongside regions. This corresponds to Blech where a struct may contain variables, instantaneous functions to invoke, and activities that can run non-instantaneous behavior when started. Compared to reactors, presented in Sec. II, methods can be invoked multiple times during an instant/reaction.



(a) UML notation (b) SCCharts notation

Fig. 2: Visual representation of a Counter class

Fig. 2a shows the UML class diagram of a Counter class. It consists of a private integer field value and three methods, to increment and decrement the counter value and a getter method to return its value. The same information is also available in the SCChart in Fig. 2b. Additionally, the given SCChart includes implementations for the three methods, illustrated by the detailed view on increment. In contrast to regions, the methods do not contain a state machine but immediate imperative code parts written in a subset of SCL [23], a synchronous subset of C. Graphically, they are displayed in gray and contain the controlflow graph (SCG [23]) representation of the SCL code. Now, consider Fig. 2a as a class interface and Fig. 2b as its implementation. Then, we can declare an instance of a counter variable, by referencing the Counter SCChart with ref Counter counter and access this object as presented later in the CounterApplication in Fig. 4c. There the three regions invoke methods counter.increment(), counter.decrement() and counter.getValue() concurrently.

Currently, like for inheritance, our compiler handles methods by macro expansion. A transformation statically expands the methods bodies into their invocation, known as function inlining. Parameters are passed by reference. Regarding compile time and code size, it might be more efficient to keep invocations. Leveraging our method of policies as scheduling interfaces, described in Sec. V, we envisage a future extension for modular compilation in which methods may remain opaque and need not be copied.

C. Automatic Diagram Synthesis

In software development there is often the problem that the actual implementation starts to diverge from the architecture defined in an earlier phase or that the architecture of an implementation needs to be documented, for example in UML diagrams. When modeling with statecharts, such as SCCharts,

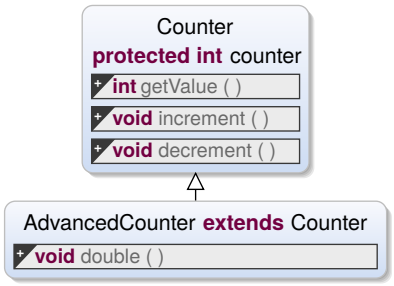


Fig. 3: Inheritance visualized

such problems are reduced by a graphical notation. The model acts as documentation and source for the generated code. However, as discussed in Sec. II, most statecharts approaches use separate diagrams to model object relations and behavior. Using the concept of transient views [20], as in SCCharts, the implemented model can be augmented or shaped into different forms to represent different aspects of a system. As a result, the tasks of designing, implementing and documenting a system start to merge while handling a single model.

Fig. 3 shows how the Counter SCCharts in Fig. 2b is extended by an additional method for doubling the counter value. Note that the internal counter variable was set to protected visibility to allow access. In this view, a generalization edge is added to visualize the inheritance relation and to provide the information usually expected of an UML class diagram in the documentation of such data structures.

IV. Object-Oriented Host Language Integration

Sec. III-B shows that SCCharts now can be used to model data structures and that static expansion creates one self-contained program that can be processed by the existing compiler without further extension of the lower-level semantics. This might suffice when SCCharts is used as the only or primary programming language in a project but in practice it is more common that SCCharts, or SLs in general, are used as a high level orchestration language. SCCharts for example are currently used by an industrial partner to replace hand-written state machines in Java and C++ projects. Such a use case requires close integration with the targeted host language and the used frameworks and other host code. On the example of SCCharts we propose a host language integration that (1) uses and supports basic OO capabilities of the host language and itself, (2) uses syntactical concepts a programmer might be familiar with from major synchronous and general-purpose languages, (3) is independent from a specific host language to allow code generation into different target platforms, and (4) provides a robust synchronous semantics, especially regarding deterministic concurrency (Sec. V).

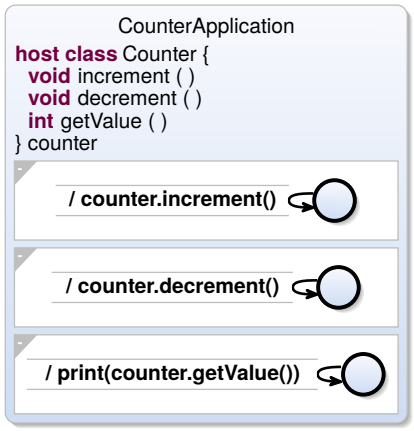
Furthermore, an OO host language integration also facilitates a more modular or incremental code generation approach, which utilizes the explicitly designed structure of the model, especially separation. An approach similar to synERJY and SOs, presented in Sec. II, interconnects separate components rather than expanding them into one monolithic program.

```
1  class Counter {
2      private int value = 0;
3
4      public void increment() {
5          value++;
6      }
7      public void decrement() {
8          value−−;
9      }
10     public void getValue() {
11         return value;
12     }
13 }
```

(a) Counter class in Java

```
1  scchart CounterApplication {
2
3    host class Counter {
4      void increment()
5      void decrement()
6      int getValue()
7    } counter
8
9    during do counter.increment()
10   during do counter.decrement()
11   during do print(counter.getValue())
12 }
```

(b) CounterApplication in textual SCCharts notation



(c) CounterApplication in graphical SCCharts notation

Fig. 4: CounterApplication modeled in SCCharts using the Java class Counter

Regarding host language integration, all established SLs, such as Esterel and Lustre, support some degree of generalized host code integration into their language, such as external function invocation and access to the host's type system. In accordance to that, SCCharts allows the declaration of external functions, such as:

extern @C "rand", @Java "Math.random" random.

Then the random function can be used in the SCChart and will be replaced by the given string variants in the generated code, depending on the targeted host language. When further compiling the generated code, the invoked methods must be (manually) provided with an implementation to allow correct linking. This is a common practice for host code integration in SLs. Additionally, variables can be declared with a specific host code type. For example, if a host implementation for the Counter class presented in Fig. 2a is available, such as the Java class in Lst. 4a, then this type can be used in SCCharts with **host** "Counter" counter.

This integration might suffice for simple function calls in C but has no concepts of OO. One could argue that, for example in the integration of the Counter, the development IDE could parse all related sources to give the modeler access to the members of that object. However, such support is not available in the current SCCharts implementation, mostly because a

certain degree of independence from the target platform is desired. Furthermore, in SCCharts as well as Esterel, Lustre and other SLs, the previously presented host integration is considered semantically unsafe, when it comes to side-effects and stateful behavior in such functions and types, see Sec. V.

We propose an OO extension to the host language integration in SCCharts. The results can be inspected in the example in Fig. 4 that uses a counter in Java as external host code in an SCChart. The Counter class in Lst. 4a implements the specification of the UML diagram in Fig. 2a w. r. t. the intended behavior. Lst. 4b shows the textual representation of the CounterApplication SCChart importing this Java class. The SCChart declares a counter variable with the external Counter class type, including the methods that should be available for this class in the SCChart. The actual behavior is defined by during actions that invoke the each method in every reaction of the model and in parallel to each other. In the graphical SCChart in Fig. 4c, which is automatically generated from the textual notation, the during actions are replaced by their equivalent state machine representation. Each of the three parallel regions contains a single state with a self transition invoking the associated method.

The new class declaration allows to encapsulate multiple declarations as members, including methods. Instead of an extern declaration the method's signature is given, as presented in Lst. 4b. Variables with this class type are statically instantiated, same as ref types referencing other SCCharts. Hence, it it possible to have multiple instances but no dynamic instantiation. The host keyword in the declaration specifies that an implementation will be available in the host language. We think this host language integration matches our goals by (1) supporting classes of the host language including member access, (2) providing Java/C++ like class syntax, and (3) retaining a certain independence from the host language, for example in object creation and memory management, regarding for example Java vs. C++. The next important aspect is a robust synchronous semantics, discussed in Sec. V.

V. Deterministic Objects

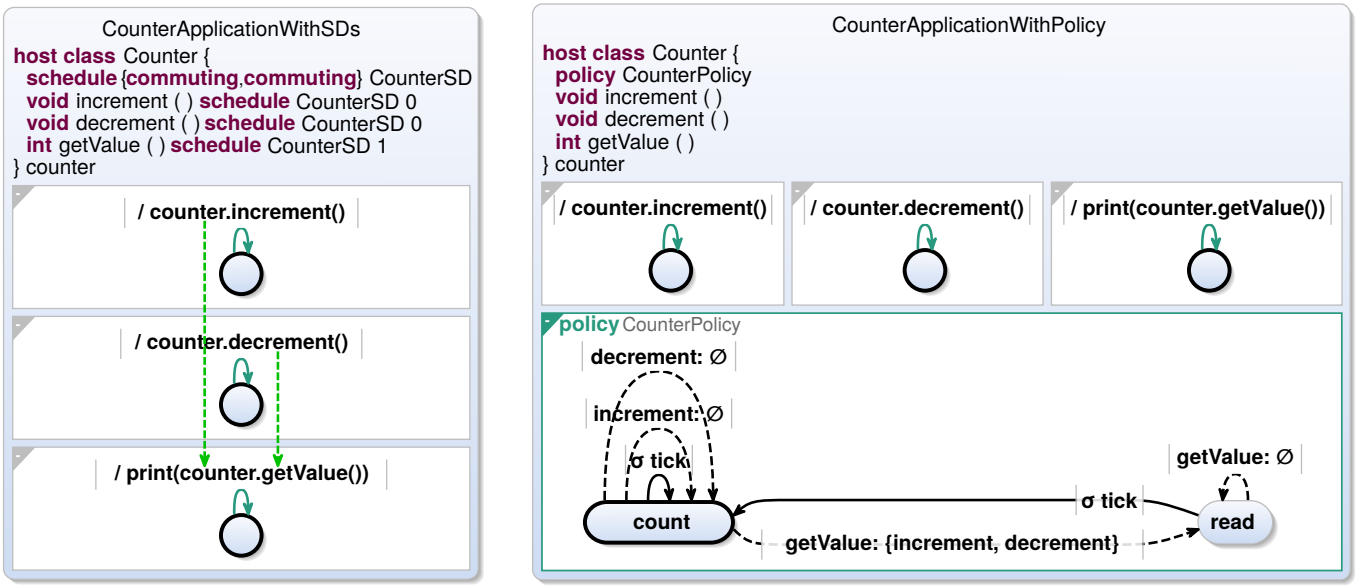
The most important feature of SLs is their deterministic concurrency, which makes them predestined for safety-critical applications. On the face of it, this seems to preclude shared data structures and thus genuine object integration. For instance, the naive compilation of the CounterApplication example in Fig. 4 renders the program vulnerable to non-determinism. The concurrent calls to increment, decrement and getValue, which access the internal counter value, are prone to *race conditions*: The return value of getValue is different depending on whether it is executed before or after an increment. Additionally, if the increment and decrement methods are not atomic, their interleaved or parallel execution may also lead to a race condition. The problem is well-known and avoided by demanding that external function calls must not have any side-effects through shared memory. Hence, to realize the CounterApplication in Fig. 4, we must code the method calls as function calls increment(&value), decrement(&value) and getValue(value) where value exposes the internal memory of the object. To ensure deterministic interaction of host code functions, all major SLs implement a strict write-before-read scheduling protocol regarding function parameters. Call-by-value parameters are considered *read* and call-by-reference parameters are *write* accesses on the respective variable. In Blech, the mutating behavior of functions arguments is controlled statically by the position of the parameter in the argument list of the function call. Since the scheduling protocol forbids concurrent writes, our CounterApplication with concurrent calls increment(&value), decrement(&value) is unschedulable and thus rejected. The problem with CounterApplication is typical for any host object integration in SLs because each method call, in general, is an implicit write access with a side-effect to the object's memory.

A solution for host integration comes from relaxing the standard notion of constructiveness by *sequential constructiveness*. It permits multiple destructive memory updates, provided these are commuting[3] with each other. This is the case for increment(&value) and decrement(&value) assuming they are atomic. As a consequence, the method calls counter.increment and counter.decrement may be classified as *(commuting) updates* and counter.getValue as a *read*. Then, the CounterApplication is sequentially constructive and schedulable under the iur protocol.

As it turns out, sequential constructiveness enables deterministic usage of host code objects without either exposing internal variables in parameters or following a white box scheduling approach and analyzing the implementation of external functions. We propose to augment classes and objects with the necessary scheduling information to avoid data races, acting as a *contract* between the synchronous program and the host object. Note that such a contract can only preserve determinism as far as its specification is correct. E.g., in case of the Counter example above this is the guarantee that increment and decrement are atomic and commuting. If supported by the host language, such contracts can also be added to objects as annotations, such as suggested by Caspi et al. [9]. We believe this approach fits well into the OO paradigm, as the existing objects or classes are extended to provide an adapted, in this case deterministic, behavior in the context of the SL. Our proposal builds on recent work on *Scheduling Directives* (SD) [22] and *Scheduling Policies* (SP) [1], which are applicable for statecharts in general. The former allow to define static indices that directly prescribe ordering of statements. The latter augment an object by an automaton that controls concurrent method calls to that object. This allows to specify a wide range of (state-dependent) access regimes.

In the following we present the Counter example from Fig. 4c, with a contract thats allows clients to invoke increment and decrement potentially multiple times in any order, but strictly before any calls to getValue. This results in a deterministic value read from a counter object in every instant.

[3] In [23] this is called "confluent" but we feel "commuting" is more precise in this context. Method execution is "confluent" because the methods are pairwise "commuting."



(a) Counter using SDs to assign scheduling indices. Resulting dependencies are visualized as green arrows.

(b) Counter augmented with an policy automaton

Fig. 5: Deterministic usage of host language object Counter in CounterApplication

A. Scheduling Directives

An SD associates a *scheduling unit*, such as a single assignment or a whole method, with a *named schedule* and an *index*. All SDs associated with the same named schedule must be scheduled according to their index, lowest index first. This induces a new schedule that may alter the pre-defined synchronization protocol of the SL. Further, the declaration of the named schedule defines the behavior if two scheduling units possess the same index. Conservatively, by default, same indices are considered conflicting. However, specific indices can be set to *commuting* meaning that the order of their execution does not matter.

Fig. 5a shows the Counter example with SDs in SCCharts. The current SCCharts compilation ensures atomicity of black box method calls and, hence, can be seen as scheduling units. The contract states that increment and decrement can be scheduled in any order, but must be scheduled before getValue. Therefore, a named schedule CounterSD with two commuting indices is declared. The index 0 is assigned to increment and decrement, meaning that their invocations can be ordered arbitrarily, but before any calls to getValue, since it has the higher index 1 assigned. Index 1 is also commuting as readers cannot conflict.

Furthermore, Esterel's valued signals can be coded using the iur protocol [19] that is at the heart of the notion of sequential constructiveness [23]. This protocol is expressible as an SD using three scheduling indices, a non-commmuting index 0 ("init") and commuting indices 1 ("update") and 2 ("read").

B. Scheduling Policies

The SDs can be generalized to SP [1], which provide even more advanced scheduling rules. SP augment an object by a *policy automaton* that controls concurrent method calls to that object such that the scheduling order can be an arbitrary precedence graph and also be state-dependent. Both the iur protocol and the static indices of SDs mentioned above are special cases.

Fig. 5b presents the CounterApplication with its associated policy automaton depicted as an SCChart in a region called CounterPolicy. The automaton has two states which capture the two different scheduling modes, before and after the first reading. Initially, in state count, all three methods calls increment, decrement and getValue are *admissible* as expressed by the dashed (instantaneous) transitions starting from count. Each transition is labeled by the name of the method call and a so-called *blocking set*, separated by a colon. Specifically, the transition labeled getValue: {increment, decrement} states that getValue is admissible but must wait for any concurrent call to increment or decrement, which take precedence. The admissible calls increment: {} and decrement: {}, on the other hand, have an empty blocking set. Hence, they are not blocked by getValue and also do not block each other. As seen in Fig. 5b, if and when the getValue is executed the automaton moves into state read. There, no increment or decrement is admissible any more, only calls to getValue. The solid (non-instantaneous) transitions labeled σ tick are the synchronous clock, which starts a new instant and resets the SP to the initial state.

An SD schedule as in Sec. V-A is the special case of a linear automaton whose states are the scheduling indices n. Each method labeled m is a transition $m : L_m$ from each state $n \leq m$ to state m. If index m is declared commuting then it is blocked only by indices larger than itself m, i.e.,

$L_m = \{k \mid k > m\}$. Otherwise, $L_m = \{k \mid k \geq m\}$. Using policy automata we can express state-dependent schedules. E.g., we could implement bounded queues in which the enqueue method is only admissible until the queue is full and the dequeue is only admissible while it is not empty. This case can be efficiently implemented and is supported by the current SCCharts compiler, the general case is likely to be computationally intractable. Other tractable state-dependent policies are pure Esterel signals [1], run-time enforcers [18], and the synchronous object policies [9].

VI. Conclusions and Outlook

We have presented approaches to introduce aspects of the OO paradigm to a synchronous statecharts dialect and extended the SCCharts language to implement our concepts. To improve structuring of large systems and to allow efficient modeling of commonalities, we introduced inheritance. In the face of the synchronous semantics and the influence of the safety critical domain in SCCharts, we decided to follow a static approach in handling this feature and to allow overriding of regions. Inheritance in combination with the new possibility for specifying and implementing methods enables modeling of complex class structures in SCCharts.

We further investigated how objects of an OO host language can be integrated into SCCharts, while retaining a deterministic behavior and the OO paradigm. We proposed to mimic the class definition of the host's objects and extend it, if necessary, by a set of rules to ensure determinism in a concurrent context. To specify a contract between the synchronous program and external objects, regarding its method invocations, we integrated two recently proposed approaches. With SDs it is possible to specify a scheduling order based on static indices, while a policy allows to model an automaton that handles precedences between method calls. Both approaches match the OO idea of extending an object by a contract and permit more flexibility than other synchronous scheduling schemes, such as used by SyncCharts or Blech.

In the future we want to investigate more OO concepts in statechart modeling, including more dynamic aspects of OO programming, as well as polymorphism, while retaining the sound semantics of SLs. Furthermore we plan to improve the convenience of handling objects in SCCharts, for example by introducing references similar to Blech. Regarding inheritance, a finer granularity for overriding behavior could further benefit the modeling process.

References

[1] Joaquín Aguado, Michael Mendler, Marc Pouzet, Partha S. Roop, and Reinhard von Hanxleden. Deterministic concurrency: A clock-synchronised shared memory approach. In *27th European Symposium on Programming, ESOP'18*, pages 86–113, Thessaloniki, Greece, April 2018.

[2] C. André, F. Boulanger, M.-A. Péraldi, J. P. Rigault, and G. Vidal-Naquet. Objects and synchronous programming. *RAIRO-APII-JESA-Journal Europeen des Systemes Automatises*, 31(3):417–432, 1997.

[3] Charles André. Computing SyncCharts reactions. *Electr. Notes Theor. Comput. Sci.*, 88:3–19, 2004.

[4] Gérard Berry. *The Esterel v5 Language Primer*, 1999. ftp://ftp-sop.inria.fr/meije/esterel/papers/primer.ps.

[5] Gérard Berry. The foundations of Esterel. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 425–454, Cambridge, MA, USA, 2000. MIT Press.

[6] Darek Biernacki, Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. Clock-directed Modular Code Generation of Synchronous Data-flow Languages. In *Proceedings of the ACM SIGPLAN/SIGBED 2008 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'08)*, pages 121–130, Tucson, AZ, USA, June 2008. ACM.

[7] Frédéric Boussinot, Guillaume Doumenc, and Jean-Bernard Stefani. Reactive objects. *Annales Des Télécommunications*, 51(9):459–473, Sep 1996.

[8] Reinhard Budde, Axel Poigné, and Karl-Heinz Sylla. synERJY An Object-oriented Synchronous Language. *Electronic Notes in Theoretical Computer Science*, 153(4):99–115, 2006.

[9] Paul Caspi, Jean-Louis Colaço, Léonard Gérard, Marc Pouzet, and Pascal Raymond. Synchronous Objects with Scheduling Policies: Introducing safe shared memory in Lustre. In *ACM Int'l Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'09)*, pages 11–20, Dublin, Ireland, June 2009. ACM.

[10] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. SCADE 6: A formal language for embedded critical software development (invited paper). In *11th International Symposium on Theoretical Aspects of Software Engineering TASE*, pages 1–11, Sophia Antipolis, France, September 2017.

[11] Derek Coleman, Fiona Hayes, and Stephen Bear. Introducing objectcharts or how to use statecharts in object-oriented design. *IEEE Transactions on Software Engineering*, 18(1):8–18, Jan 1992.

[12] Friedrich Gretz and Franz-Josef Grosch. Blech, imperative synchronous programming! In *2018 Forum on Specification Design Languages (FDL)*, pages 5–16, September 2018.

[13] Olivier Hainque, Laurent Pautet, Yann Le Biannic, and Eric Nassor. Cronos: A separate compilation toolset for modular Esterel applications. In *World Congress on Formal Methods*, volume 1709 of *LNCS*, pages 1836–1853. Springer, September 1999.

[14] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, Sept. 1991.

[15] David Harel and Eran Gery. Executable object modeling with statecharts. In *Proceedings of the 18th International Conference on Software Engineering*, ICSE '96, pages 246–257. IEEE Computer Society, 1996.

[16] David Harel and Hillel Kugler. *The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML)*, pages 325–354. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[17] Marten Lohstroh, Martin Schoeberl, Andres Goens, Armin Wasicek, Christopher Gill, Marjan Sirjani, and Edward A. Lee. Invited: Actors revisited for time-critical systems. In *56th ACM/IEEE Design Automation Conference (DAC)*, June 2019.

[18] Srinivas Pinisetty, Partha S. Roop, Steven Smyth, Stavros Tripakis, and Reinhard von Hanxleden. Runtime enforcement of cyber-physical systems. *ACM Transactions on Embedded Computing Systems, Special Issue for ESWEEK/EMSOFT '17*, 16(5s):178:1–178:25, 2017.

[19] Karsten Rathlev, Steven Smyth, Christian Motika, Reinhard von Hanxleden, and Michael Mendler. SCEst: Sequentially Constructive Esterel. In *Proceedings of the 13th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE '15)*, Austin, TX, USA, September 2015.

[20] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. Just model! – Putting automatic synthesis of node-link-diagrams into practice. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '13)*, pages 75–82, San Jose, CA, USA, September 2013.

[21] Bran Selic. Real-time object-oriented modeling. *IFAC Proceedings Volumes*, 29(5):1 – 6, 1996. IFAC Workshop on real Time Programming WRTP 96, Gramado, Brazil, 4-6 November.

[22] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. Practical causality handling for synchronous languages. In *Proc. Design, Automation and Test in Europe Conference (DATE '19)*, Florence, Italy, March 2019. IEEE.

[23] 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. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, pages 372–383, Edinburgh, UK, June 2014. ACM.