Toward Object-oriented Modeling in SCCharts

ALEXANDER SCHULZ-ROSENGARTEN and STEVEN SMYTH, Kiel University, Germany
MICHAEL MENDLER, Bamberg University, Germany

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.

CCS Concepts: • **Software and its engineering** → **Model-driven software engineering**; *Abstraction, modeling and modularity*; *Object oriented languages*; **Inheritance**; **Classes and objects**; *Concurrent programming structures*; *Visual languages*; Orchestration languages; • **Computer systems organization** → Embedded software;

Additional Key Words and Phrases: Synchronous languages, object orientation, inheritance, determinacy, state machine modeling

ACM Reference format:
Alexander Schulz-Rosengarten, Steven Smyth, and Michael Mendler. 2021. Toward Object-oriented Modeling in SCCharts. *ACM Trans. Embed. Comput. Syst.* 20, 4, Article 37 (May 2021), 26 pages.
https://doi.org/10.1145/3453482

1 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

Authors' addresses: A. Schulz-Rosengarten and S. Smyth, Department of Computer Science, Kiel University, Olshausenstr. 40, 24098, Kiel, Germany; emails: {als, ssm}@informatik.uni-kiel.de; M. Mendler, Faculty of Information Systems and Applied Computer Sciences, Bamberg University, An der Weberei 5, 96047, Bamberg, Germany; email: michael.mendler@uni-bamberg.de.

1539-9087/2021/05-ART37 $15.00
https://doi.org/10.1145/3453482

Unified Modeling Language (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. Even if the methodology of OO is sometimes controversially discussed, the concepts have proven to be successful [4]. Hence, it is compelling to try and exploit the benefits of OO also in a specialized domain such as embedded and safety-critical systems. This domain typically involves complex interactions between system components and the environment, while imposing stringent requirements on functional correctness, real-time performance and fault tolerance. Some dynamic aspects of OO, such as runtime polymorphism, might seem unfit for such scenarios. However, a language like Ada [2] shows that OO is still feasible in this domain. In the presence of strict typing and static verification, it is possible to prevent vulnerabilities and ensure real-time behavior. This facilitates the use in complex applications, such as autonomous car control systems [27].

Synchronous languages (SL) are likewise designed for the programming of safety-critical embedded systems. Synchronous languages 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.

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 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 experience in Ada and other programming domains, OO has a lot to offer here. Also, object-based modeling has already long been recognized as a useful structuring principle in intermediate languages for the modular compilation of traditional SLs [11, 20]. 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 synchronous programming by OO facilities, as far as they fit. The objective of this article is to lay the foundation for OO in SLs using **Sequentially Constructive Statecharts (SCCharts)** as an example language, since it combines modern model-driven engineering with a powerful synchronous semantics. This provides the benefits of OO modeling while remaining on safe semantical terrain. Object-oriented features are compiled conservatively by basic semantic transformations that can be inspected and verified on source level, thus grounding their semantics in the existing well-established execution model of SCCharts. With this foundation more dynamic aspects can be investigated and included into SLs in the future.

1.1 Brief Introduction to Synchronous Languages and SCCharts

Synchronous Languages provide built-in concurrency with deterministic well-defined semantics. Concurrency in general is often a challenge and can compromise a safety-critical system by introducing *race conditions* [24]. SLs solve this problem traditionally by two techniques. First, 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* where the programs perform a reaction to the current state of its environment. Second, 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

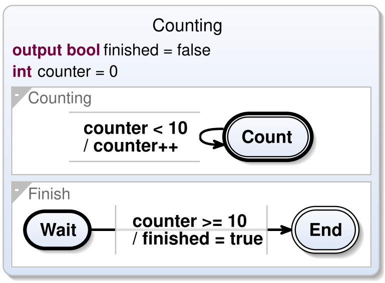


Fig. 1. Simple SCCharts example that counts to 10.

a static *causality analysis* to establish that a program is schedulable, i.e., is satisfying the synchronization protocol for each memory reference. Programs are considered *constructive* if a scheduling order can be found. Non-constructive programs are rejected.

Synchronous languages come in different programming styles and with slightly different synchronization protocols. The most prominent SLs are Esterel [10] and Lustre [21]. Lustre is based on dataflow equations and is commercially used by the Safety-Critical Application Development Environment [16] that allows the graphical modeling of dataflow diagrams. In Lustre and its derivatives communication is done via single-writer/multi-reader (1-place) channels. Esterel supports an imperative coding style and uses multi-writer/multi-reader signals. The intra-instant synchronization of Esterel implements a *write-before-read* protocol. It first schedules all writers, The same applies to Lustre, except that a combination function is not needed, since there is only one writer per channel. However, Esterel is more expressive, since the concurrent multi-writer model supports *reaction to absence*, which is not available in Lustre. Both Esterel and Lustre share the limitation of the *write-before-read* protocol, which prohibits a shared signal or channel to be overwritten during an instant. Destructive updates must be coded in *thread-local variables* that cannot be shared concurrently.

The **Sequentially Constructive (SC)** model of computation [39] relaxes this rather strict protocol, unifying signals, channels and local variables 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. It also supports reaction to absence, so that Esterel signals can be modeled as a special instance of SC-variables.

SCCharts [39] are a dialect of Harel's statecharts with SC semantics. They are inspired by SyncCharts [5], a similar statecharts notation but with the synchronous semantics of Esterel. Figure 1 illustrates a simple SCChart that waits 10 instants by counting them and then sets the finished output and terminates. The root state Counting has two variables declared, the Boolean output finished that is initially false, and the local variable counter initialized to zero. The behavior is modeled in two concurrent regions. Region Counting has a single state, that is the initial (bold border) and final state (double border) of that region, with a self-transition. This transition has a triggering condition (counter $<$ 10) and an effect (counter++). The solid line indicates that this transitions is delayed, meaning that the statechart has to remain for at least one instant in the source state before the transition can be taken. The counterpart are immediate transitions, displayed as dashed lines, that can leave a state in the same instant it is entered. Hence, except in the first instant, this region will increment the counter in each instant until it reaches 10. In classical SLs that do not have SC semantics, testing the counter before incrementing it would be rejected as non-constructive, since the trigger and the effect are treated as concurrent that results in a causality cycle under the read-before-write protocol. However, with SC the sequential program-order is taken into account, which resolves the data dependency between the effect and the trigger. Hence, the program is *sequentially constructive*. The second region Finish stays in its initial state Wait until the counter reaches 10. SCCharts provide instantaneous concurrent communication, hence the final state End will be entered in the same instant as the Counting region increments the counter. The iur protocol is used to find a deterministic scheduling order and orders the reading of counter in region Finish after its write access in the Counting region. When state End is entered, both regions reach their final state and the program terminates.

Similarly to other SLs, SCCharts is defined with a minimal set of *core* language features that ease the static analysis and facilitate code generation. Additionally, it provides many *extended* features, that can be considered syntactical sugar. These are translated into equivalent structures built with core features by performing model-to-model transformations in the compilation of SCCharts [26]. For example, a single during action could be used to replace the entire region Counting. Specifically, during **if** counter < 10 do counter++ would be transformed into the same region (except the naming) as in Figure 1, without the need to explicitly model the region, state, and transition.

Contributions and Outline

We first discuss related work in Section 2 and then present the following contributions[1]:

- We present the first statechart dialect that permits OO modeling under the principles of SLs. It is built as an extension of SCCharts providing *inheritance* (Section 3.1) and *class-based data structures* (Section 3.2) to facilitate abstraction and reusability. The benefits and limitations of OO design in SCCharts are discussed in Section 3.3 and the integration into the model-driven engineering tool KIELER is described in Section 3.4.
- We propose mechanisms to ensure the determinism of host language objects under the shared memory concurrency of SLs while retaining their encapsulation under a black box scheduling approach. In Section 4 we explain the integration of class-based host data structures from a syntactic and code generation perspective. Section 5 presents mechanisms for determinism based on *scheduling directives* (Section 5.1) and *scheduling policies* (Section 5.2) and discuss how these can enrich classes for subtype polymorphism (Section 5.3).

We conclude in Section 6 and give an outlook on future work in Section 7.

2 RELATED WORK

In the context of embedded systems, Ada [2] is the most notable OO language. It provides standardized imperative programming with a strong type system and non-deterministic concurrency with locking mechanism for synchronization, while facilitating verifiability through a design-by-contract methodology. OO features include encapsulation, generics, and subtyping with dynamic dispatch. In contrast to Ada, SCCharts focuses on model-driven engineering and uses the synchronous principle for deterministic concurrency.

Concepts from OO have also been adapted into statechart dialects. *ObjectCharts* [17] by Coleman et al. characterize the communication behavior of state machines as objects 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. Following on from ObjectCharts, Harel and Gery [22] present *O-charts* to specify classes and structures and use statecharts for the modeling of the object's behavior. They introduce an OO statecharts version, which resulted in the *Rhapsody*

[1]This article extends an earlier publication in the Forum on Specification and Design Languages (FDL 2019) [32].

semantics of statecharts [23] 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 or modifying transitions. IBM's Rational Rhapsody provides similar ways of refining statecharts.[2] It is possible to change triggers and actions, as well as the target of transitions but not the source. Changes in the topology of states, by reparenting, is not possible either. Such a fine-grained way of adjusting the inherited state machine also requires tool support for modeling across inheritance relations. Model elements can be set to *inherited*, propagating changes in super classes to its sub classes, *overridden*, ignoring changes in super classes but still synchronizing deletion of states, or *regular* where the sub class is decoupled from its super implementation. Syriani et al. propose a more restricted set of rules for the refinement of statecharts to ease static verification [38]. For inheritance in SCCharts, we use a simplified approach and allow overriding only on regions. This requires less restrictions and synchronization effort as regions are completely replaced when overridden, corresponding to the regular mode in Rhapsody. *ROOMcharts* [33] is another statecharts dialect, used in the **real-time object-orented 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.

The above OO statechart dialects inspired the OO modeling concepts introduced to SCCharts. However, we do not follow the typical approach of separating the modeling of the system's structure from the modeling of behavior, as will be discussed in Section 3.4. This encourages a uniform semantics-oriented modeling as it is characteristic for SLs. Furthermore, in contrast to the above OO statechart dialects, SCCharts reconcile behavioral decomposition and concurrency with determinism and static verifiability.

There are many SLs that adapt some OO concepts into their language and semantics. André et al. [6] introduce **synchronous objects (SO)** based on the *reactive object model* [12]. This approach divides the program into a collection of **regular host code objects (RO)** and SO that communicate with each other. Messaging allows SO to communicate instantaneously and preserve the synchronous semantics. The resulting directed interconnection graph is required to be acyclic, because modules are considered black boxes that cannot interleave with each other. Communication with ROs is done via signals that can be read outside SO but inputs to SO require special handling by *interface objects* to enter the synchronous messaging mechanism. The structure of a system using SO is represented by an object-modeling technique class diagram augmented by communication interfaces. André et al. support SyncCharts, among other SLs, for specifying the internal behavior of an SO. The SyncCharts dialect is not extended by OO features, in contrast to the SCCharts presented here, 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 as we do here.

The synchronous dataflow languages Lustre has been extended by OO-like syntax for shared-memory modularization [15]. *Scheduling policies* are used to expose and verify the execution order of dataflow equations packaged in object structures. Our class policies for SCCharts play a similar role for the SC scheduling semantics that, unlike Reference [15], permits destructive memory updates.

The synchronous OO language synERJY [13] 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 use signals to communicate with each other

[2]https://www.ibm.com/support/knowledgecenter/SSB2MU_8.2.0/com.ibm.rhp.uml.diagrams.doc/topics/rhp_c_dm_stchrt_inheritance.html.

and 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 similar to our **Scheduling Directives (SDs)**, which will be discussed in Section 5.1. However, it does not include a combination with graphical modeling or ways to deterministically include host code objects. Inheritance and subtyping is only implemented in predefined types.

The imperative SL language Blech [18], recently introduced in an industrial context, provides C++-like abstract data types that can be instantiated like classes and shared by concurrent threads. The methods to access the data objects are normal procedure calls with call-by-reference parameters. The determinism is guaranteed by the *write-before-read* protocol, which is generalized for nested and composite memory structures, in contrast to Lustre or Esterel, which only permit atomic signals or scalar dataflow variables to be shared concurrently. Blech permits sequential memory updates on shared objects within an instant, like in SCCharts. Causality analysis is facilitated by labeling all methods statically as *mutating* or *non-mutating* (on the self-object) and all method arguments as *mutable* or *non-mutable*. In this way, the compiler can abstract the body of methods for modular (black box) scheduling or for integrating host code. In addition, all object references are static. Hence, potential write-write conflicts on overlapping objects (e.g., by aliasing) are statically resolvable. The *write-before-read* protocol of Blech forces a rather rigid form of scheduling that is not as flexible as our scheduling directives or as expressive as our policies discussed in Section 5. Specifically, it does not permit concurrent writing to the same atomic memory cell, and hence, it does not support reaction to absence, Esterel signals, or shared counter structures as used in Section 3.2 below. Also, Blech does not support inheritance.

Lohstroh et al. [25] present a deterministic real-time refinement of the actor model, called *reactors*. It uses OO principles for scopes of nested reactors with the aim eventually to include inheritance, subtyping, and generic programming and to provide libraries with generic reactors, such as parallel scatter-gather patterns.

3 OBJECT-ORIENTED MODELING IN SCCHARTS

Following the definition of Wegner [40], OO languages distinguish themselves from object-based languages by providing classes and inheritance. Classes are templates for constructing objects and inheritance establishes relations that allow reusing or altering code of a parent of the implementation. In the presence of a type system, inheritance often also expresses subtype relations, where an object is allowed to substitute objects of its super classes or interfaces. With this polymorphism, a method call needs to be bound to the actual implementation based on the type of the involved object and the type of the parameters (in case of overloaded methods). There are many powerful static analyses, especially in functional OO languages, that resolve polymorphism at compile time. This early binding enables method inlining and other optimizations. However, in the face of unconstrained runtime-mutable object pointers, as present in general purpose OO languages, the actual implementation of a method can only be determined at runtime, requiring dynamic binding. Without static type checking, it is not even ensured that a called method exists in an object. Dynamic binding is an important aspect of OO languages [4] but at the same time it should be considered carefully, due to the necessity of a lookup operation at each method invocation. In the safety-critical domain and for embedded and real-time systems, this dynamic aspect can be problematic. For SLs, whose main feature is ruling out programs that are non-deterministic, this poses the challenge of statically analyzing all possible runtime substitutions of a type and data accesses at compile-time to rule out race conditions. This also includes dynamic memory allocation and alias management when working with objects and mutable references.

Nonetheless, we are convinced that including OO features in SLs can greatly improve their expressiveness, effectiveness and convenience when it comes to modeling. In this work we focus on introducing OO to SCCharts. At this point, we aim to implement our OO concepts in a conservative manner and exclude features that may cause runtime overheads or jeopardize static code analysis, such as runtime polymorphism. We build upon well-established and tested compilation mechanisms of SCCharts without breaking existing concepts and with an overhead that is as small as possible. Even without dynamic aspects, supporting inheritance (Section 3.1) and classes (Section 3.2) in the modeling with SLs can help dealing with large and complex systems, especially in combination with modern modeling techniques (see Section 3.4). It can also be seen as an alternative to procedural decomposition of models as in classical SLs. This can help developers used to OO design and UML class diagrams and changes the cognitive perception of a synchronous statechart model. Section 3.3 elaborates further on the implemented OO features and the effects of OO modeling with SCCharts compared to a classical approach.

Based on the new work reported here, the SCCharts language can be further extended with subtyping and generic programming principles. It also allows to handle polymorphism deterministically for SLs, which leads into a similar direction as *behavioral subtyping*. Section 5.3 gives an idea how our concept for deterministic host code objects can facilitate this. However, subtyping and polymorphism will be subject to future work and is only sketched in Section 7.

3.1 Inheritance

Inheritance is a core OO concept to express commonalities between entities of a system. It allows (for class-based OO) to base the class for an object upon another class. This reuses, alters or extends existing behavior (implementation) to create a new kind of objects adjusted to their purpose.

Reusing code in SLs, such as Esterel [9], SyncCharts or Lustre, is traditionally handled by a macro expansion mechanism. A program is split up into several self-contained modules, which then can reference other modules and rebind their interface variables. These references are statically expanded at compile time, similar to function inlining or macros of the C preprocessor. This mechanism fits well with SLs, as it eases static analysis of programs and is semantically solid. SCCharts also provide such a macro expansion mechanism: A state can reference another SCChart and rebind its inputs and output variables. However, redefining common variables in each module again and binding them at each use are tedious tasks and do not facilitate a smooth modeling process. SCCharts were used in a number of student projects in the last years [34]. Especially for larger projects, such as modeling a controller for a model railway installation with multiple different trains, we found that expressing commonalities between modules improves the design structure. Similarly, experience from our collaboration with industry partners suggests the usefulness of common default behavior for states with the option of adjusting it in instantiation contexts. To address this issue, we improve the expressiveness of SCCharts and make the modeling process more similar to design principles known from modern (usually OO) programming languages. Hence, we decided to extend SCCharts further toward OO and to introduce *inheritance*.

Inheritance in SCCharts now allows to derive states from one or more *base states*[3] 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 with inheritance support fine-grained altering of states and transitions. However, here we follow a more conservative approach and allow only *region overriding*. This corresponds more to common OO programming languages, such as Java or C++, where methods are the units that are subject to overriding.

[3]The term "superstate," following super class, might be more obvious here. However, in SCCharts a superstate is already defined as a state that contains inner behavior such as regions [39].

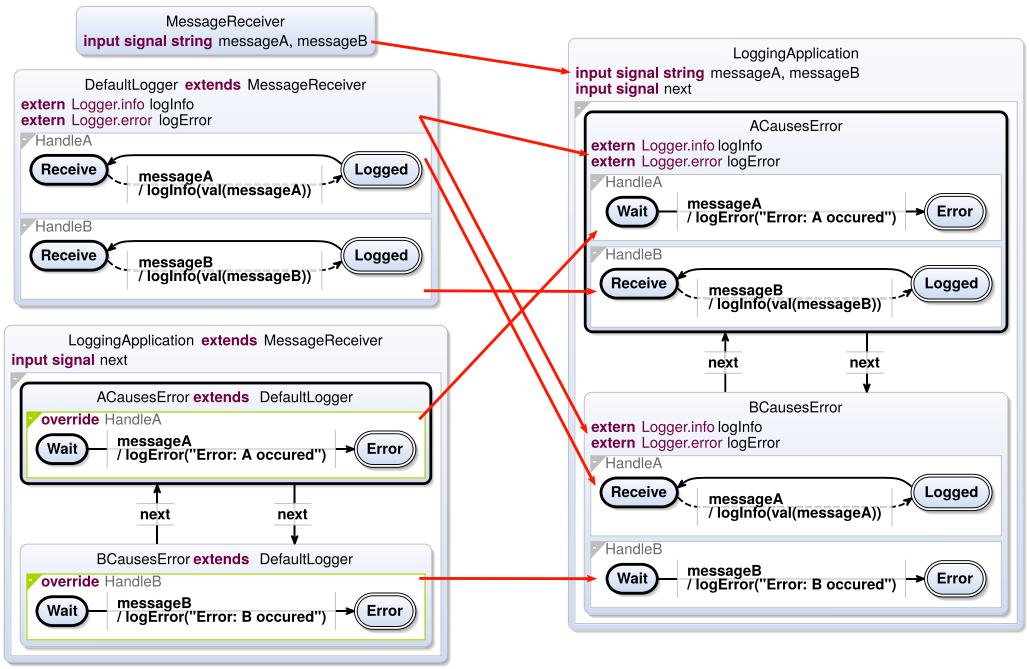

MessageReceiver
input signal string messageA, messageB
DefaultLogger extends MessageReceiver
extern Logger.info logInfo
extern Logger.error logError
HandleA
Receive
messageA / logInfo(val(messageA))
Logged
HandleB
Receive
messageB / logInfo(val(messageB))
Logged
LoggingApplication extends MessageReceiver
input signal next
ACausesError extends DefaultLogger
override HandleA
Wait
messageA / logError("Error: A occured")
Error
next
next
BCausesError extends DefaultLogger
override HandleB
Wait
messageB / logError("Error: B occured")
Error
LoggingApplication
input signal string messageA, messageB
input signal next
ACausesError
extern Logger.info logInfo
extern Logger.error logError
HandleA
Wait
messageA / logError("Error: A occured")
Error
HandleB
Receive
messageB / logInfo(val(messageB))
Logged
next
next
BCausesError
extern Logger.info logInfo
extern Logger.error logError
HandleA
Receive
messageA / logInfo(val(messageA))
Logged
HandleB
Wait
messageB / logError("Error: B occured")
Error

Fig. 2. 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.

Figure 2 shows on the left-hand side how inheritance can be used in SCCharts. The SCChart is a simplified model of a real-world example.[4] In the underlying scenario, incoming messages, here reduced to 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, then 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 [39] in SCCharts and removed by a model-to-model transformation in the first step of the compilation. Figure 2 presents on the right-hand side the result of this transformation. We perform a macro expansion with finer granularity and static dispatch for overriding of regions. All variables and regions are copied into their extending states

[4]Our industrial partner from the railway domain uses SCCharts to replace hand-written state machine code by models and generated code. In the context of a C++ project, the developers found the need for states to have common default behaviors. An example is the described logging of messages. It is only reasonable to address such a use case by means of OO, especially since C++ developers are already used to this methodology.

w.r.t. overriding. The red arrows in the figure 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.

If the inheritance hierarchy of a state is cyclic, then it cannot be statically expanded and the compiler rejects the model. Furthermore, the support of multiple inheritance, by allowing more than one base state, introduces challenges such as the diamond problem. If base states contribute variables or regions with the same id but defined in different states, then we require that the region must be overridden to define a single behavior for that element or the model is rejected (as for conflicting variables, since they have no overriding support). This allows us to statically handle this feature. It corresponds to the strategy for default methods in Java 8 interfaces.

Furthermore, we extend the variable scoping of SCCharts and add access modifiers for variables and regions to allow visibility restrictions. When expanding inheritance, variables with restricted visibility (e.g., private) are renamed to prevent name clashes with variables in the extending scope. We also support accessing the scope of the base state while overriding, by using the super keyword, as known from languages such as Java.

We have implemented inheritance in the open-source KIELER tool[5] that provides the reference implementation for SCCharts and also facilitates the modeling and understanding of inheritance in SCCharts, see Section 3.4.

3.2 Class Modeling

Traditionally, SCCharts are primarily used to model a system as a statechart, such as the logger example in Figure 2. 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 stateful behavior when started. Accordingly, methods in SCCharts are currently restricted to perform only instantaneous behavior, i.e., they must not contain any synchronous delay (pause) as this would introduce states.

Figure 3(a) shows the UML class diagram of a Counter class. It consists of a private integer field counter and three methods, to increment and decrement the counter and a getter method to return its value. The same information is also available in the SCChart in Figure 3(b). 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, written in a subset of SCL [39], a synchronous subset of C. Graphically, they are displayed in gray and contain the controlflow graph (SCG [39]) representation of the SCL code.

Currently, like for inheritance, our compiler handles methods by macro expansion. The default transformation statically expands the method bodies into their invocation site, known as function inlining. Parameters are passed by reference. Regarding compile time and code size, it might be more efficient to keep invocations. Hence, there is also an alternative transformation that synthesizes code with unexpanded methods. However, in this approach we consider each method invocation atomic, since we do not want to split up method bodies into multiple methods for interleaving. As a consequence, this restricts the number of SCCharts programs that can be

[5]https://rtsys.informatik.uni-kiel.de/kieler.

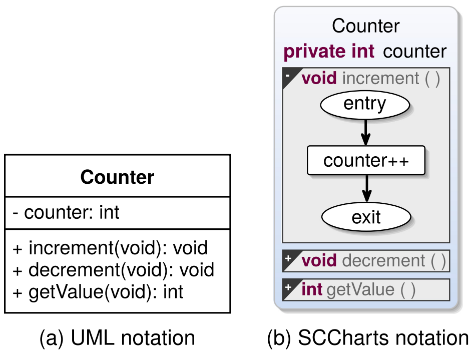


Fig. 3. Visual representation of a Counter class.

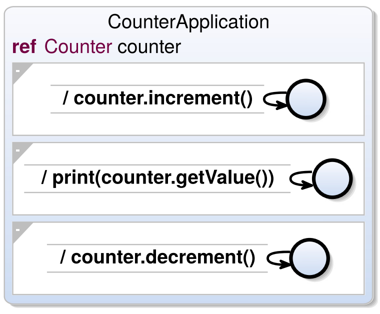


Fig. 4. CounterApplication SCChart using Counter class.

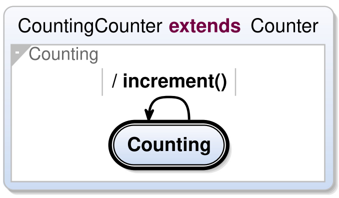
accepted. For without interleaving of concurrent method bodies, the compiler might not be able to schedule the shared memory accesses into a deterministic order. If this is unacceptable, then efficient "gray-box" techniques can be used to preserve the causality information of the method body [29]. Leveraging our method of policies as scheduling interfaces, described in Section 5, we envisage a future extension for modular compilation in which methods may remain opaque.

Now, consider Figure 3(a) as a class interface and Figure 3(b) as its implementation. In our extended SCCharts, the Counter SCChart can act as a class type for variables. The SCChart in Figure 4 declares an instance in the `counter` variable by referencing the Counter SCChart with **ref** Counter counter. The three regions invoke the methods counter.increment(), counter.decrement(), and counter.getValue() concurrently in every instant. The SCCharts semantics prescribes a scheduling that orders the printing of the value after the other concurrent method invocations. The consequence is that the counter value always stays zero. This may not look reasonable, but the semantics of this example will be used in Section 5 to illustrate scheduling on host objects and has no further implications on the example in this context.

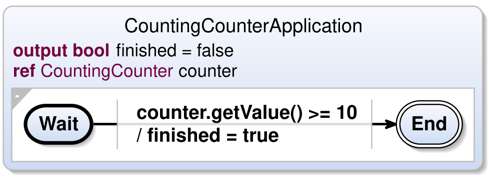
There are fields and methods in SCCharts to design classes as usual in OO design. However, SCCharts used as classes can still have regions. Such classes implement inherently active behavior that runs alongside the regular program. For example, a class with its own worker thread that processes an input queue or a simple state machine that tracks the inner state of the object. Figure 5(a) presents a simple scenario based on the counter example. The CountingCounter extends the Counter SCChart, in such a way that it will autonomously increment the counter in every instant. This is done by adding the region Counting containing a state with a self-transition that invokes `increment`.

To illustrate how SCCharts-based classes are translated, Figure 5(b) presents the SCChart CountingCounterApplication that creates a variable `counter` based on the CountingCounter. The SCChart simply waits until 10 instants have passed and then terminates. This is done by a transition from the initial state to the final state triggered when the counter value reaches 10.

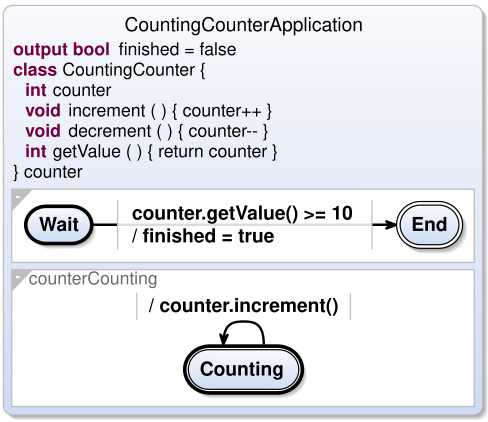
Similarly to inheritance, SCCharts-based classes are not part of the kernel language but an extended feature that is replaced statically in the first steps of the compilation. Figure 5(c) shows the result of this translation step. First, the inheritance hierarchy of CountingCounter is statically expanded, as described in Section 3.1. Then the referenced type is translated into a native class declaration. They allow a more classical, not model-based, approach for defining classes, strongly inspired by the Java syntax. They are mainly designed to provide an integration of objects and classes from the host languages into SCCharts, as described in Section 4. In this case, they are used



(a) CountingCounter extending Counter by automatic increments


CountingCounterApplication
output bool finished = false
ref CountingCounter counter
Wait
counter.getValue() >= 10
/ finished = true
End

(b) Use of CountingCounter class


CountingCounterApplication
output bool finished = false
class CountingCounter {
int counter
void increment ( ) { counter++ }
void decrement ( ) { counter-- }
int getValue ( ) { return counter }
} counter
Wait
counter.getValue() >= 10
/ finished = true
End
counterCounting
/ counter.increment()
Counting

(c) Handling of referenced SCCharts types

Fig. 5. Example for using and handling of regions in class design with SCCharts.

to bring SCCharts into a form that can be easily represented by a host language class. Such a class declaration may only contain variables and methods.

In Figure 5(c), the class CountingCounter contains the counter variable and the three implemented methods, taken from the SCChart. The region Counting is directly added to the CountingCounterApplication SCChart, since it now contains the class declaration. The region is renamed to counterCounting to allow multiple instances of such a class. The method call counter.increment is also adjusted to refer to the instance variable.

Transforming regions of SCCharts-based classes like this is possible, since we decided to statically instantiate variables of class types. We want to circumvent the need to adapt to different memory management mechanisms in target languages, such as C/C++ and Java. Hence, memory for object variables is allocated globally once at program start or can be allocated and deallocated based on the scope and lifetime of the local variable together with its state or region. However, this is handled by the generated code and not the user. Furthermore, we currently do not allow assignments on object variables. Hence, each such variable can be considered a constant pointer to the data structure. In the future, we want to extend SCCharts to support references similar to the capabilities of Blech (Section 7), but we first decided for a more conservative approach as references may easily compromise the deterministic semantics of SCCharts. As a consequence, this currently limits the ability to use designs that pass and store objects with different, yet compatible, subtypes. We here lay a foundation that can enable such designs in the future, to a limit where determinism can still be guaranteed. Note that classes from a host language might require different handling in memory allocation (Section 4) and do not automatically ensure determinism, which requires special scheduling (Section 5).

Table 1. Object-oriented Characteristics in SCCharts

Feature	Current Support/Restriction
objects	supported (Section 3.2)
- composition	class-based, defined classical or as SCChart
- creation/destruction	only static, only read-only object variables
- encapsulation	private/protected/public visibility
inheritance	supported (Section 3.1)
- relations	multiple inheritance, no virtual inheritance, static expansion
- override	only for regions, planned for methods
subtyping	future work (Section 7)
- parametric polymorphism	planned
- dispatching	planned, static with policies (Section 5.3)

With SCCharts now supporting methods and regions to specify instantaneous imperative functionality and non-instantaneous stateful behavior for objects, it shares some similarities with Blech. Methods correspond to functions and regions resemble activities. The main difference to activities is that regions are not directly parameterized, statically instantiated (singletons for each instance), and automatically start with the SCCharts programs.

3.3 Benefits and Limitations of Object Orientation in SCCharts

With the introduction of classes, methods, and inheritance with overriding, SCCharts now supports important features of an OO language. Table 1 gives an overview of the current OO features and those that will be included in the future. As initially described the dynamic runtime aspects of OO are challenging to adapt to the safety-critical and embedded domain. We decided to conservatively focus the presented work and the current implementation in SCCharts on static analyzability, memory boundedness and good predictability for execution time. Consequently, objects are currently restricted to static instantiation and read-only instance variables, which rules out runtime polymorphism and dynamic dispatching. Nonetheless, some restrictions can be partially lifted in the future based on extensions of the concepts presented here, as Section 5.3 and Section 7 describe.

Including OO in SCCharts also allows a different methodology of designing SCCharts. To compare this OO approach with the traditional way of modeling SCCharts, we need to consider that modularity and reusability is classically achieved by module expansion. If we wanted to model the LoggingApplication in Figure 2 with reusable modules, then each of the different regions for handling logging behavior (logInfo vs. logError) would require its own SCChart. Only this way, they could be referenced and correctly composed in the two modes of LoggingApplication to achieve the same behavior. This is a more procedural approach to decomposing the problem compared to the OO design. The more important difference is that there is no longer a notion of a *default behavior* for logging, inherent to each state. The reason is that the module concept in its basic form cannot express common default behavior and classically only allows to parametrize input data rather than behavior. Of course, procedural parameters in combination with default values or special forms of function overloading could be introduced to tackle this problem and allow generalizable and adjustable behavior. Otherwise, the developer has to circumvent comment default behavior by modeling fine- grained modules that are composed explicitly in each instance. However, we think the OO paradigm is a more natural and well-established way to approach this use case. The result is a more expressive language with generalization and parameterization capabilities due to inheritance and overloading.

In terms of performance, our approach of transforming inheritance early on in the compilation comes at no real cost compared to classical macro expansion. We modeled the LoggingApplication, all previously presented Counter examples, and a small robot controller both in the new OO design and in the classical way using SCCharts modules. When expanding modules and inheritance respectively, there are only minimal differences. They are mainly caused by fact that SCCharts modules can only reference states and thus require an additional hierarchy level, whereas inheritance allows to override regions directly. The support for classes and methods also resolves naturally by inlining and produces very similar results. In our tests we could not identify any indication that our approach is more costly than classical modeling. Instead, we see great potential in a code generation that does not transform away the OO characteristics, as in our currently quite conservative approach, but reflects these structures in an OO target language to improve code size and performance.

Regarding the time effort needed to model SCCharts using the OO methodology, we likewise have not experienced any significant drawback over a procedural approach. However, to get a reliable evaluation of the effects of OO in SCCharts on the design time, a larger user study is necessary, which we consider future work.

To test the presented OO concepts in a small case study, we modeled a controller for the steam boiler example [1]. It is a well-known model for cyber-physical systems. In the scenario, a control program maintains a safe water level in a steam boiler by communicating with its physical devices, such as multiple pumps and sensors for throughput, water and steam. The program has to be able to detect different device and operation failures and react by switching the operation mode until the devices are repaired or issue an emergency stop. Steam boiler controllers have been implemented in several languages. For example, Büssow and Weber [14] use an OO approach to decompose the problem and derive an *architectural view*. In their implementation they use classical statecharts and the specification language Z. In our SCCharts model we identified very similar entities and inheritance relations between components. The entire controller is too large to be presented in full. Figure 6 shows selected SCCharts and their inheritance relations, visualized as additional generalization edges. It focuses on the components that best utilize the new OO features. Most notable is the common failure and repair protocol that is specified for most physical units. We implement it abstractly in the RepairHandling region of the PhysicalUnit SCChart. It is extended by the different controllers for the sensors and actuators, each of which handles failures specific to the unit. The general process of reporting the failure, awaiting an acknowledgment and then the repair message is inherited from the base SCCharts and bound to the device-specific communication messages. The AbstractFailiure interface is separated from the PhysicalUnit, because there is the MonitoredPump that is not a physical unit itself. It has no repair protocol itself but accumulates the Pump and PumpControl and signals a failure if one of these fails. Another commonality is modeled in the AbstractWaterLevel, that is able to compute and return the abstract water levels, such as *normal* or *critical*, based on the current level. It is extended by the WaterSensor that communicates with the actual device to retrieve the level. Whereas, Computed-WaterLevel is used to replace the WaterSensor if it failed and the program is in the *rescue* mode. It computes the water level based on the throughput at the pump controls and steam sensor. In the design by Büssow and Weber [14], they also model an abstract physical unit and aggregate the pump and pump control into a monitored pump. They do not specify a water level independent from the sensor, but they describe that a more fine-grained decomposition could further improve modularity.

The specification of the steam boiler problem itself does not assume any specific design paradigm. Hence, there are alternative implementations of this problem that do not have such an architecture and achieve the same functionality. However, our example and the version by Büssow and

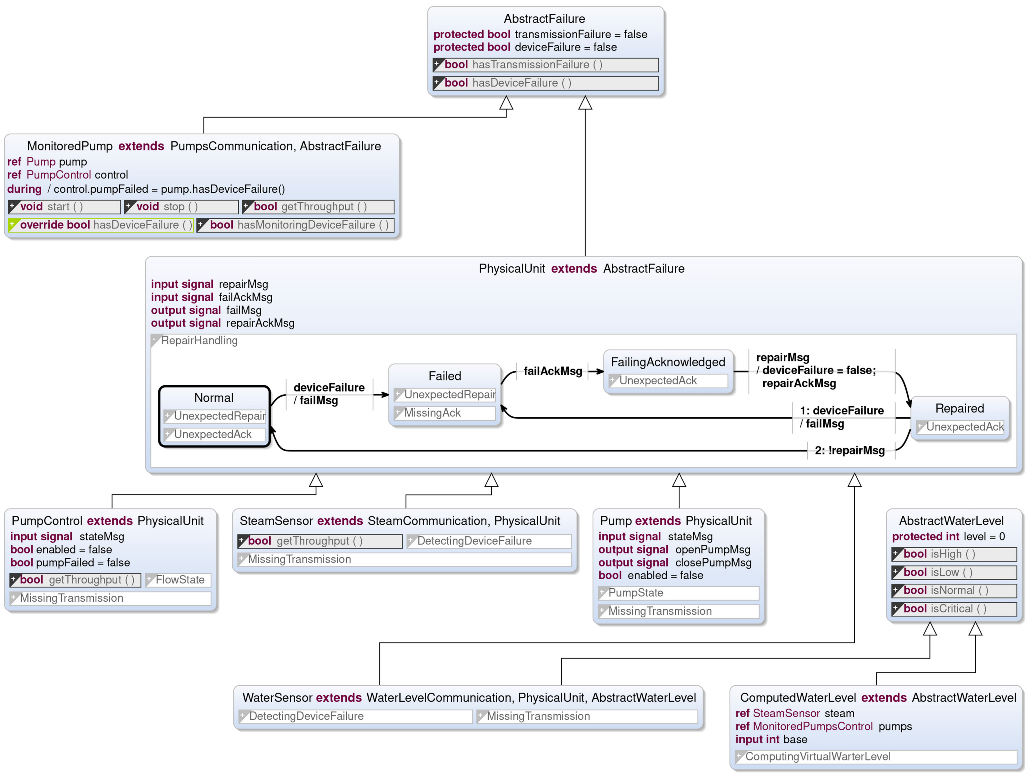

AbstractFailure
protected bool transmissionFailure = false
protected bool deviceFailure = false
bool hasTransmissionFailure ( )
bool hasDeviceFailure ( )
MonitoredPump extends PumpsCommunication, AbstractFailure
ref Pump pump
ref PumpControl control
during / control.pumpFailed = pump.hasDeviceFailure()
void start ( )
void stop ( )
bool getThroughput ( )
override bool hasDeviceFailure ( )
bool hasMonitoringDeviceFailure ( )
PhysicalUnit extends AbstractFailure
input signal repairMsg
input signal failAckMsg
output signal failMsg
output signal repairAckMsg
RepairHandling
Normal
UnexpectedRepair
UnexpectedAck
deviceFailure / failMsg
Failed
UnexpectedRepair
MissingAck
failAckMsg
FailingAcknowledged
UnexpectedAck
repairMsg / deviceFailure = false; repairAckMsg
1: deviceFailure / failMsg
2: !repairMsg
Repaired
UnexpectedAck
PumpControl extends PhysicalUnit
input signal stateMsg
bool enabled = false
bool pumpFailed = false
bool getThroughput ( )
FlowState
MissingTransmission
SteamSensor extends SteamCommunication, PhysicalUnit
bool getThroughput ( )
DetectingDeviceFailure
MissingTransmission
Pump extends PhysicalUnit
input signal stateMsg
output signal openPumpMsg
output signal closePumpMsg
bool enabled = false
PumpState
MissingTransmission
AbstractWaterLevel
protected int level = 0
bool isHigh ( )
bool isLow ( )
bool isNormal ( )
bool isCritical ( )
WaterSensor extends WaterLevelCommunication, PhysicalUnit, AbstractWaterLevel
DetectingDeviceFailure
MissingTransmission
ComputedWaterLevel extends AbstractWaterLevel
ref SteamSensor steam
ref MonitoredPumpsControl pumps
input int base
ComputingVirtualWarterLevel

Fig. 6. Selection of SCCharts used in the steam boiler controller and their inheritance relations.

Weber illustrate that the steam boiler fits into the well-established methodology of OO software design. With our approach in SCCharts we are able to naturally express OO architecture and its implementation in one model. Furthermore, it has already been demonstrated elsewhere that such a design can facilitate modeling for embedded systems, see for instance Mouelhi et al. [27] who present an OO design for autonomous control systems based on Ada.

3.4 Automatic Diagram Synthesis and KIELER

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, such problems are reduced by a graphical notation. The model acts as documentation and source for the generated code. However, as discussed in Section 2, most statecharts approaches use separate diagrams to model object relations and behavior. Using the concept of *transient views* [31] 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.

The open-source KIELER tool implements this concept of transient views for SCCharts. Languages available in KIELER, such as SCCharts, are implemented with the Xtext framework[6] that allows to create textual domain-specific languages for models. These models are then visualized in

[6]https://www.eclipse.org/Xtext/.

graphical views with the KLighD framework [31] that uses automatic layout[7] for arranging these diagrams.

Textual editing with automatically generated graphical views combines many features that facilitate the modeling experience, especially for OO SCCharts. The on-demand visualization of models as diagrams can be configured and adjusted in various ways to serve the developer's needs and assist in the modeling process. The generalization edges in Figure 6 are such an example. They provide the information usually expected of a UML class diagram in the documentation of such a project. However, in this case the entire program structure can be interactively explored in varying granularity, down to the implementation level, by expanding and collapsing regions and filtering details. With an interactive model-based compiler [36], as in KIELER, the intermediate steps of the compilation also become inspectable. This allows the user to get a deeper understanding on how the compilation actually translates a program and enables the user to inspect and verify each step, which, e.g., facilitates debugging. Similarly to Figure 2, the source models and the expanded model can be shown side by side. There are also views dedicated to detecting causality issues or improving the understanding of the generated code [35].

At the same time, there is still a textual variant available to offer the benefits of modern editors and source control. For example, syntax highlighting, jump-to-declaration, and content-assist. Furthermore, the textual syntax is closer to OO programming in major languages, such as Java and C++, which lowers the entry burden for new developers. Compared to O-charts or Rhapsody where class relations are modeled separately from the behavior implementation, SCCharts has a single source language with all relations automatically inferred and configurable visualized.

4 OBJECT-ORIENTED HOST LANGUAGE INTEGRATION

Section 3.2 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 a SL is used as the only or primary programming language in a project. However, in practice it is more common that SLs are used as a high-level orchestration languages. SCCharts, for example, are currently employed 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, frameworks, and other host code. For SLs 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 is 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 (Section 5).

Furthermore, the ability to integrate elements of an OO host language in a likewise structured model, facilitates a more modular or incremental code generation, which can utilize the explicitly designed structure of the model, especially separation. An approach similar to synERJY and SOs, presented in Section 2, interconnects separate components rather than expanding them into one monolithic program.

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

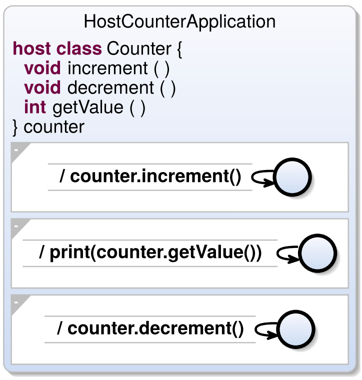
[7]https://www.eclipse.org/elk/.

```
class Counter {
  private int value = 0;

  public void increment() {
    value++;
  }
  public void decrement() {
    value−−;
  }
  public int getValue() {
    return value;
  }
}
```

(a) Counter class in Java

```
scchart HostCounterApplication {
  host class Counter {
    void increment()
    void decrement()
    int getValue()
  } counter

  during do counter.increment()
  during do counter.decrement()
  during do print(counter.
   getValue())
}
```

(b) HostCounterApplication in textual SCCharts notation



(c) HostCounterApplication in graphical SCCharts notation (with expanded during actions)

Fig. 7. HostCounterApplication modeled in SCCharts using the Java class Counter.

variants in the generated code, depending on the targeted host language. Later, when compiling the generated code, the invoked methods must be (manually) linked with an implementation. 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 Figure 3(a) is available, such as the Java class in Figure 7(a), then this type can be used in SCCharts using the declaration notation "**host** "Counter" counter".

This integration might suffice for simple function calls in C but has no concepts of OO. One could argue, for example in the integration of the Counter, that the integrated development environment (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 traditional host integration is considered semantically unsafe, when it comes to side effects and stateful behavior in such functions and types, see Section 5.

We propose an OO extension to the host language integration in SCCharts that is based on class declarations, as introduced in Section 3.2. The results can be inspected in the example in Figure 7 that adapts the CounterApplication in Figure 4 by using the Counter implemented in Java as external host code. The Counter class in Figure 7(a) implements the specification of the UML diagram in Figure 3(a). Figure 7(b) shows the textual representation of the HostCounterApplication SCChart importing this Java class. The SCChart declares a counter variable with the external host class Counter specification, including the method signatures that must be available for this class. The behavior of this SCChart is defined by during actions that invoke each method in every reaction instant of the model and concurrent to each other. In the graphical SCChart in Figure 7(c), which is automatically generated from the textual notation, the during actions are replaced by their equivalent state machine representation. It corresponds to the SCChart in Figure 4.

The host class declaration allows to define available class members, including methods, as presented in Figure 7(b). Variables with this class type are statically instantiated, same as ref types referencing other SCCharts. Hence, it is 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. This is the main difference to non-host class declarations introduced by the user or by ref types. They need to be synthesized appropriately when generating code, further discussed in Section 4.1.

```
#hostcode "import java.util.List;"
#hostcode "import java.util.ArrayList;"

scchart JavaList {
  input int value

  host class "List<Integer>" {
    bool add(int value)
    int size()
  } list = `new ArrayList<Integer>(50)`

  during if value > 0
      do list.add(value); print(list.size())
}
```

(a) SCChart using an `ArrayList` from the Java Collections framework.

```
typedef struct {
  struct Counter {
    int counter;
  } counter;
  int _counter_getValue_return;
  char _GO;
  char _TERM;
} TickData;

void logic(TickData *d) {
  if (!d->_GO) {
    d->counter.counter++;
    d->counter.counter--;
    d->_counter_getValue_return = d->
        counter.counter;
    printf("%d", d->
        _counter_getValue_return);
  }
}

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

void tick(TickData *d) {
  logic(d);
  d->_GO = 0;
}
```

(b) Generated C code for the CounterApplication example using the netlist-based approach.

Fig. 8. Examples for host language specific classes and code generation.

We think this host language integration matches our goals by (1) supporting classes of the host language including member access, (2) providing common Java/C++ like class syntax, and (3) retaining a certain independence from the host language. This concerns the language of SCCharts itself, which is uncoupled from different strategies in object creation and memory management, for example with Java vs. C++, and allows to support various target languages in code generation as described in Section 4.1. The last aspect (4) is a robust synchronous semantics, discussed in Section 5.

4.1 Remarks on Class Declarations in SCCharts and Code Generation

The new class declarations allow us to integrate host classes into SCCharts and enable the user to define own classes that are directly translated into the generated code. The code generation for SCCharts in the KIELER tool currently supports two main target languages, C as a low-level language for embedded devices and Java as a platform independent and more high-level language. This implementation is extended by class declarations. For host classes, which can be expected to be available in the host language, the code generation is fairly simple as the objects can be used directly. Sometimes it might be necessary to add host code specific annotations in the SCCharts to ensure the compiler imports the class correctly. Figure 8(a) shows an SCChart that uses a Java `ArrayList`. There are special `hostcode` annotations to import the correct classes and the host class declaration uses host code specific object creation. Due to natural differences in programming languages and their approaches to memory management, we favor the concept of statically instantiated classes in SCCharts. For flexibility, we still allow to adjust this behavior, since in this case a default instantiation with `new List()` would not work. As a consequence, this SCChart can only be compiled into Java code. The SCChart JavaList declares and uses only the `add` and `size` method of the Java `List` interface. In each instant where the input `value` is greater than zero, it is added to the list and the new size is printed. To prevent non-deterministic behavior of such host code objects, the techniques discussed in Section 5 can be used. This example illustrates how SCCharts achieve a certain independence from host languages w.r.t. to the syntax. It is possible to integrate specific host language types and classes but the same languages constructs can be used to integrate objects from C++, Python or JavaScript if supported by a code generation in the compilation back-end of SCCharts. The SCCharts language itself does not need to be adjusted to these languages. This *polyglot* nature is also present in other orchestration languages for embedded systems, such as Reactors [25].

When synthesizing code for classes declared in SCCharts, the code generation might have to adjust to the capabilities of the host language. For Java, code generation is fairly straight-forward, due to the syntactical similarities. In C, a struct can be used to represent a class, since the inlining of methods removes them from the classes beforehand, as discussed in Section 3.2. If methods are not inlined, then they need to be defined separately and the struct needs to be passed to the function to allow access to the object members. Figure 8(b) shows the generated C code for the CounterApplication in Figure 4 using the netlist-based compilation approach [39]. Without going into much detail about this approach and the structure of the code, the struct representing the Counter class is located in the TickData struct that holds the state of the statechart. In the logic function, which performs a single step of the statechart, the inlined method bodies manipulate and read the counter value in every but the first instant indicated by !GO.

5 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 inhibits genuine object integration. The CounterApplication in Figure 4 uses an object with methods defined in SCCharts. Hence, everything in the program is subject to the SCCharts semantics checked by a static analysis for SC schedulability [39]. According to this semantics, the concurrent regions will be scheduled into a deterministic order such that increment and decrement will be executed before getValue. However, this is only possible since the analysis can use a white box approach for the entire program. When using host objects, as in the HostCounterApplication example in Figure 7, the implementation is usually not available when the synchronous program is analyzed. This is a general problem and renders the program vulnerable to non-determinism. Without the knowledge about memory accesses in methods, the regions of the HostCounterApplication could be ordered randomly. 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/decrement. Additionally, if the increment and decrement methods are not atomic, then their interleaved or parallel execution may also lead to a race condition. Note that for SCCharts, in the absence of data dependencies induced by memory access, the scheduling falls back to the syntactical order in the source code. However, this still yields a different behavior of the program, since the value is printed before it is decremented.

The issue of non-determinism with black box function calls is well-known to SLs. It is usually avoided by demanding that external functions must not have any side effects through shared memory. Hence, to realize the HostCounterApplication in Figure 7(c), 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 such host code functions, most 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 function 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 HostCounterApplication with concurrent calls increment(&value), decrement(&value) is unschedulable and thus rejected. The problem with HostCounterApplication is typical for OO host integration in SLs, because each method call, by default, has a side effect on the object's memory. This corresponds to the basic OO principle of *encapsulation.* Blech supports side effects on objects by marking the functions mutating but forbids the concurrent invocation of these functions, consequently rejecting HostCounterApplication as well.

A solution for host integration comes from relaxing the standard notion of constructiveness by *sequential constructiveness*, as in SCCharts. It permits multiple destructive memory updates, provided these are commuting[8] 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 HostCounterApplication is sequentially constructive and schedulable under the iur protocol.

As it turns out, sequential constructiveness facilitates 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. The program context uses the contract as a restriction for the admissible scheduling of host method calls and the host code guarantees memory determinism under all admissible schedules. If supported by the host language, then such contracts can also be added to objects as annotations, such as suggested by Caspi et al. [15]. This approach fits well into the OO paradigm. Scheduling contracts may act as interface types extending the existing objects or classes to provide deterministic behavior in the context of SLs. Our proposal builds on recent work on SD [37] and **Scheduling Policies (SP)** [3], which are applicable for statecharts in general. The former allow to define static indices that directly prescribe the 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.

For illustration, we present the HostCounterApplication example from Figure 7(c) in the following. It is extended by a contract that 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 and corresponds to the semantics under a white box iur scheduling.

5.1 Scheduling Directives

An SD associates a *scheduling unit*, such as a single assignment or a whole method, with a *named schedule* and an *index.*[9] 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 predefined synchronization protocol of the SL. Conservatively, scheduling units with the same indices are considered *conflicting* by default. Indices can be set to *commuting* if the order of their execution does not matter. For example, as said before, it is common in SLs to order variable accesses according to a strict *write-before-read* regime. An SD can be used to override this scheduling order, e.g., to assign new values to variables *after* they have been read concurrently. For black box host code SDs are a way of defining non-trivial schedules beyond the simple conventions such as call-by-value/call-by-reference discussed above.

Figure 9 shows the HostCounterApplication example with SDs in SCCharts. The current SCCharts compilation ensures atomicity of black box method calls that, therefore, can be seen

[8]In Reference [39], this is called "confluent," but we feel "commuting" is more precise in this context. Method execution is "confluent," because the methods are pairwise "commuting."

[9]We here avoid the term "priority" to avoid confusion with the priorities of priority-based scheduling [39], where the highest priority is executed first.

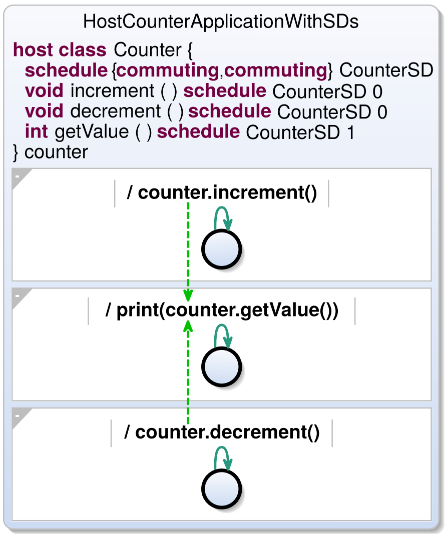

HostCounterApplicationWithSDs
host class Counter {
schedule {commuting,commuting} CounterSD
void increment ( ) schedule CounterSD 0
void decrement ( ) schedule CounterSD 0
int getValue ( ) schedule CounterSD 1
} counter
/ counter.increment()
/ print(counter.getValue())
/ counter.decrement()

Fig. 9. Deterministic usage of host language object Counter in HostCounterApplication by using to assign scheduling indices. Resulting dependencies are visualized as green arrows.

as scheduling units. Our aim is to state 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, which has assigned the higher index 1. Note that it is reasonable to set index 1 also commuting, which enables multiple readers to call getValue concurrently, but strictly after all updates occurred.

Esterel's *valued signals* can be coded using the iur protocol [30] that is at the heart of the notion of sequential constructiveness [39]. This protocol is also expressible as an SD using three scheduling indices, a noncommuting index 0 ("init") and commuting indices 1 ("update") and 2 ("read"). Hence, it is possible to create access schedules, which are common for SLs, also for black box host code objects. Moreover, this approach provides a more flexible and powerful way of influencing scheduling compared to predefined regimes in SLs. Even for Blech, which allows functions to mutate objects and exposes read and write accesses on parameters, SDs give the programmer the opportunity to write deterministic programs that would be rejected by the Blech compiler. Such an example is HostCounterApplication: Since increment and decrement are both mutating they could not be invoked concurrently in Blech. Assigning scheduling indices with SDs follows the same principle as the manual precedences for conflicting writers in synERJY [13].

5.2 Scheduling Policies

The SDs can be generalized to SPs [3], which provide even more advanced scheduling rules. SPs 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. The iur protocol and the static indices of SDs mentioned above are special cases.

Figure 10 presents the HostCounterApplication with its associated policy automaton depicted as an SCChart in a region called CounterPolicy. The automaton has two states, count and read, 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

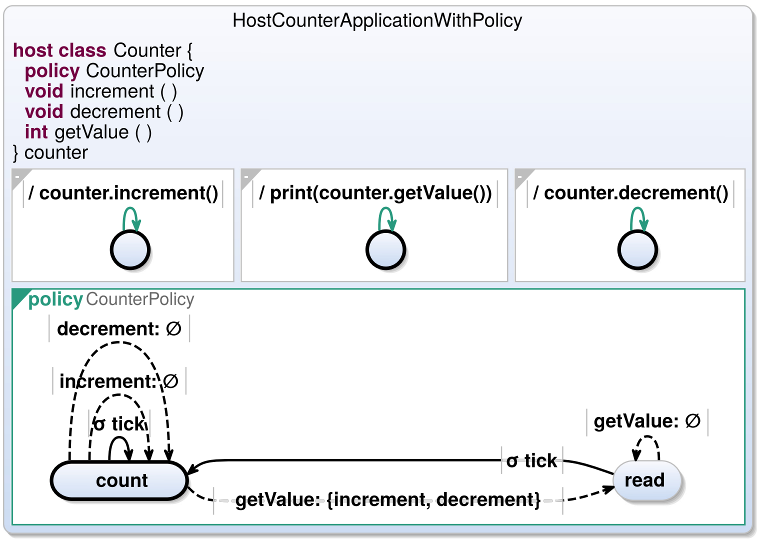


Fig. 10. Deterministic usage of host language object Counter in HostCounterApplication by augmenting it with an policy automaton.

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: ∅, however, have an empty blocking set. Hence, they are not blocked by getValue and also do not block each other. As seen in Figure 10, when 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 count mode.

SD schedules (Section 5.1) are special cases of single-state SP automata. Each method labeled m is a self-loop $m : L_m$ on the only policy state with a blocking set L_m. If index m is declared commuting, then it is blocked only by indices smaller than itself m, i.e., $L_m = \{k \mid k < m\}$. Otherwise, $L_m = \{k \mid k \leq m\}$. Using general policy automata we can express state-dependent schedules. For example, 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. Other tractable state-dependent policies are pure Esterel signals [3], runtime enforcers [28], and the synchronous object policies [15].

5.3 Deterministic Polymorphism

SPs, as well as SDs, specify custom scheduling regimes that can replace or augment the computational model of a language. Such user/library-defined scheduling rules allow us to achieve determinism while adhering to OO principles [19]. When an SCCharts class is defined with a custom SD/SP scheduling contract then each (static or dynamic) instance of the class guarantees determinism (*memory coherence* [3]) of the method calls for all policy-conformant schedules, including concurrent object accesses. Policy contracts generate static or dynamic schedules depending on whether the class typing and method binding can be resolved statically or only at runtime. Using the notions of *policy interfaces* and *interface extensions*, as recently proposed by Gretz et al. [19], we briefly sketch how SD/SP contracts can be leveraged for polymorphism and subtyping. The central invariant to be preserved is that well-typed programs exhibit deterministic behavior that is functionally dependent on controllable input alone rather than unobservable internal scheduling choices.

The semantics for *synchronous activities* [19] permits black box procedural abstraction under the SC model of computation. The scheduling of black box procedures is controlled by precedence policies so as to preserve memory coherence and determinism at runtime. At any moment in the execution, the store is protected by a *memory interface* $C = (O, \pi)$ where O is the set of accessible and typed object paths and π is a *precedence policy* that specifies which qualified methods $m \in Mtd[O]$ are *admissible* along the paths O and how these must be ordered in a C*-conformant* schedule. If π does not specify any precedence between two admissible methods, then they are *concurrent independent* (commuting) and can be scheduled in any order. Such precedence policies [19] are a superset of SDs [37] and correspond to state-less SP [3]. Note that the memory interface C may be static or change at runtime as new objects are instantiated and deallocated.

The idea is to treat a method call $\mathrm{o}.m(\mathrm{o}_1, \mathrm{o}_2, \ldots, \mathrm{o}_n)$ that passes objects o_i to method m in object o like a black box procedure call with call-by-reference parameters. When the method m is resolved, statically or via dynamic dispatch tables, a memory interface $C_m = (O_m, \pi_m)$ is retrieved that over-approximates a prediction of the memory accesses in the method's body for the current synchronous instant. The interface C of the current store at the call site and the formal interface C_m of the called method $m \in Mtd[O]$ provide sufficient information for safe scheduling [19]. Each method call $\mathrm{o}.m(\mathrm{o}_1, \mathrm{o}_2, \ldots, \mathrm{o}_n)$ with actual parameter objects o_i induces a path map $f_m : Mtd[O_m] \rightarrow Mtd[O]$ that translates the formal method calls of the generic method to the associated actual method calls in the current memory. Memory aliasing is captured by the fact that f_m is not injective in general. The sets O_m and O must be predictive (static or dynamic) over-approximations of locations potentially accessible during the current instant, also considering any indirections via object pointers. For the method call to be *memory safe*, the path map f_m must preserve the path types associated with O_m and preserve admissibility and concurrent independence as expressed in π_m. The former is the standard type checking problem of OO. The latter is the extension of type checking for memory coherence and determinacy. Specifically, it requires that if two admissible methods a, b are concurrently independent under π_m, then their instantiations $f_m(a), f_m(b)$ must be concurrently independent in π. This ensures that the instantiated method code respects the scheduling precedences required by the current store and does not introduce non-determinacy. Path maps f satisfying the above induce an *extension* $f : C_1 \sqsubseteq C_2$ of interfaces that correspond to a subtyping relation: Every store that is coherent (i.e., behaves deterministically) under C_2 is also coherent for C_1 under the (aliasing) map f. A program that is deadlock-free under C_1 will also be deadlock-free under the more generous C_2 and in a C_2-coherent store and exhibit the same behavior as when scheduled under the more restrictive interface C_1. For safety-critical applications it will be important that the applicable policy interfaces and aliasing maps at all call sites can be determined statically. This must be ensured by syntactic restrictions limiting the flexibility of OO programming style, such as banning mutable object references. Programs can thus be *locally optimized* by shifting along the $\sqsubseteq$ relation in the spectrum between purely sequential and purely concurrent schedules while preserving program behavior.

6 SUMMARY

We have presented approaches to introduce some aspects of the OO paradigm to a synchronous statecharts dialect and extended the SCCharts language to implement our concepts. To improve

```
scchart GenericState
 <T extends DataInterface,
  P extends AbstractProcessor> {
 private ref T data

 initial state main is P<T>(data)
 //...
}
```

Fig. 11. A generic SCChart mockup.

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 presented how these modeled classes are translated and discussed the role of transient views in the development process.

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 scheduling directives SDs it is possible to specify a scheduling order based on static indices, while a scheduling policy SPs allows to model state-dependent 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. Finally, we have sketched how (state-less) precedence policies can be leveraged to induce a form of subtype polymorphism.

7 OUTLOOK

In this work, we focused on introducing basic OO features and principles in the context of synchronous statecharts that lay the foundation for future developments toward OO and their evaluation with SCCharts. OO aspects, such as class instances and methods are handled conservatively here to ease their conformance with the synchronous semantics and application for the embedded and safety-critical domain. We plan to continue this cautious path not to overload SCCharts with semantically unwieldy dynamic OO concepts, such as dynamic dispatch for regions and methods.

Nonetheless, we want to improve the convenience and expressiveness of objects in SCCharts. For example by improving the support for abstract classes and interfaces. With static instantiation and read-only references on classes, there is currently no opportunity for subtyping and polymorphism. Parameterized states would give the user more possibilities to generalize the design of the system and reuse code. Figure 11 shows a mockup from our ongoing work of a generic SCChart that has two type parameters. `T` is the type for the data that should be processed and `P` is an abstract SCChart for processing such data. The mockup shows how `T` could be used to declare a `data` variable and `P` is instantiated via regular macro expansion but bound to the `data` variable and type `T`. Such a feature could be handled statically and fits well into the OO concepts introduced so far. In particular, the `extends` relation on classes corresponds to interface extension $\sqsubseteq$ in the sense of Section 5.3. If the code for a state $\texttt{main}(\texttt{x} : \texttt{DataInterface}) : \texttt{P}$ is deterministic and deadlock-free, then $\texttt{main}(\texttt{data} : \texttt{T})$ will also be deadlock-free and deterministic provided $\texttt{DataInterface} \sqsubseteq \texttt{T}$. The argument $\texttt{data} : \texttt{T}$ supports all methods of `DataInterface` under no more scheduling constraints as specified by `DataInterface`. Moreover, if the methods of `data` are functionally equivalent to those in `x`, and `data` implements `T` coherently, then the behavior of `main(x : DataInterface) : P` and `main(data : T) : P` is identical. Similarly, if `AbstractProcessor` $\sqsubseteq$ `P`, then the state `main(data : T) : P` can be substituted for `s : AbstractProcessor` in any scheduling context. This preserves determinacy and deadlock-freeness if the code of `main(x : DataInterface) : P` is sound for the interface specifications `DataInterface` and `P`. The crucial next step for the development of sound semantics of inheritance and subtyping will be the integration of precedence policies [19] into the existing interface theories for SLs such as developed by Pouzet et al. [29] and Benveniste et al. [8]. A related open problem is to define a compositional semantics for SCCharts modules that is compatible with the policy-based operational semantics [3] and permits us to provide semantical invariants preserved under inheritance. A promising setting for such a theory might be the general notion of contracts proposed by Benveniste et al. [7].

Furthermore, with OO there are new opportunities for a more modular code generation. For example the approach of André et al. [6] could be used to synthesize SOs from SCCharts modules.

ACKNOWLEDGMENT

The authors thank the FDL and Synchron workshop community for their feedback and inspiring discussions. We are especially grateful for the suggestions of Albert Benveniste and Benoit Caillaud regarding the relation of synchronous semantics and OO. We also thank the anonymous reviewers for their valuable comments.

REFERENCES

[1] Jean-Raymond Abrial. 1996. *Steam-Boiler Control Specification Problem.* Springer, Berlin, 500–509. DOI: https://doi.org/10.1007/BFb0027252

[2] AdaCore. 2016. High-Integrity Object-Oriented Programming in Ada, v1.4. Retrieved from http://extranet.eu.adacore.com/articles/HighIntegrityAda.pdf.

[3] Joaquín Aguado, Michael Mendler, Marc Pouzet, Partha S. Roop, and Reinhard von Hanxleden. 2018. Deterministic concurrency: A clock-synchronised shared memory approach. In *Proceedings of the 27th European Symposium on Programming (ESOP'18).* 86–113. DOI: https://doi.org/10.1007/978-3-319-89884-1_4

[4] Jonathan Aldrich. 2013. The power of interoperability: Why objects are inevitable. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!'13).* Association for Computing Machinery, New York, NY, 101–116. DOI: https://doi.org/10.1145/2509578.2514738

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

[6] Charles André, Frédéric Boulanger, Marie-Agnès Péraldi, Jean-Paul Rigault, and Guy Vidal-Naquet. 1997. Objects and synchronous programming. *J. Eur. Syst. Automat.* 31, 3 (1997), 417–432.

[7] Albert Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Thomas Henzinger, and Kim Guldstrand Larsen. 2018. *Contracts for System Design.* Now, Foundations and Trends.

[8] Albert Benveniste, Benoît Caillaud, and Jean-Baptiste Raclet. 2012. Application of interface theories to the separate compilation of synchronous programs. In *Proceedings of the IEEE Conference on Decision and Control (CDC'12).* 7252–7258.

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

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

[11] Darek Biernacki, Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. 2008. 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).* ACM, 121–130.

[12] Frédéric Boussinot, Guillaume Doumenc, and Jean-Bernard Stefani. 1996. Reactive objects. *Ann. Télécommun.* 51, 9 (Sep. 1996), 459–473. DOI: https://doi.org/10.1007/BF02997708

[13] Reinhard Budde, Axel Poigné, and Karl-Heinz Sylla. 2006. synERJY an object-oriented synchronous language. *Electr. Notes Theor. Comput. Sci.* 153, 4 (2006), 99–115.

[14] Robert Büssow and Matthias Weber. 1996. A steam-boiler control specification with statecharts and Z. In *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control.*Springer-Verlag, Berlin, 109–128.

[15] Paul Caspi, Jean-Louis Colaço, Léonard Gérard, Marc Pouzet, and Pascal Raymond. 2009. Synchronous objects with scheduling policies: Introducing safe shared memory in Lustre. In *Proceedings of the ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'09).* ACM,11–20.

[16] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. 2017. SCADE 6: A formal language for embedded critical software development (invited paper). In *Proceedings of the 11th International Symposium on Theoretical Aspects of Software Engineering (TASE'17).* Sophia Antipolis, France, 1–11.

[17] Derek Coleman, Fiona Hayes, and Stephen Bear. 1992. Introducing objectcharts or how to use statecharts in object-oriented design. *IEEE Trans. Softw. Eng.* 18, 1 (Jan. 1992), 8–18. DOI:https://doi.org/10.1109/32.120312

[18] Friedrich Gretz and Franz-Josef Grosch. 2018. Blech, imperative synchronous programming!. In *Proceedings of the Forum on Specification Design Languages (FDL'18).* 5–16. DOI:https://doi.org/10.1109/FDL.2018.8524036

[19] Friedrich Gretz, Franz-Josef Grosch, Michael Mendler, and Stephan Scheele. 2020. Synchronized shared memory and procedural abstraction: Towards a formal semantics of Blech. In *Proceedings of the Forum on Specification and Design Languages (FDL'20).* DOI:https://doi.org/10.1109/FDL50818.2020.9232942

[20] Olivier Hainque, Laurent Pautet, Yann Le Biannic, and Eric Nassor. 1999. Cronos: A separate compilation toolset for modular Esterel applications. In *Proceedings of the World Congress on Formal Methods*, Lecture Notes in Computer Science, Vol. 1709. Springer, 1836–1853.

[21] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (Sep. 1991), 1305–1320.

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

[23] David Harel and Hillel Kugler. 2004. *The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML).* Springer, Berlin, 325–354. DOI:https://doi.org/10.1007/978-3-540-27863-4_19

[24] Edward A. Lee. 2006. The problem with threads. *IEEE Comput.* 39, 5 (2006), 33–42.

[25] Marten Lohstroh, Martin Schoeberl, Andres Goens, Armin Wasicek, Christopher Gill, Marjan Sirjani, and Edward A. Lee. 2019. Invited: Actors revisited for time-critical systems. In *Proceedings of the 56th ACM/IEEE Design Automation Conference (DAC'19).*

[26] Christian Motika, Steven Smyth, and Reinhard von Hanxleden. 2014. Compiling SCCharts—A case-study on interactive model-based compilation. In *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'14)*, Lecture Notes in Computer Science, Vol. 8802. 461–480. DOI:https://doi.org/10.1007/978-3-662-45234-9

[27] Sebti Mouelhi, Daniela Cancila, and Amar Ramdane-Cherif. 2017. Distributed object-oriented design of autonomous control systems for connected vehicle platoons. In *Proceedings of the 2017 22nd International Conference on Engineering of Complex Computer Systems (ICECCS'17).* 40–49. DOI:https://doi.org/10.1109/ICECCS.2017.32

[28] Srinivas Pinisetty, Partha S. Roop, Steven Smyth, Stavros Tripakis, and Reinhard von Hanxleden. 2017. Runtime enforcement of cyber-physical systems. *ACM Trans. Embed. Comput. Syst.* 16, 5s (2017), 178:1–178:25.

[29] Marc Pouzet and Pascal Raymond. 2010. Modular static scheduling of synchronous data-flow networks—An efficient symbolic representation. *Des. Autom. Emb. Syst.* 14, 3 (2010), 165–192.

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

[31] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. 2013. 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).* IEEE, 75–82. DOI:https://doi.org/10.1109/VLHCC.2013.6645246

[32] Alexander Schulz-Rosengarten, Steven Smyth, and Michael Mendler. 2019. Towards object-oriented modeling in SCCharts. In *Proceedings of the Forum on Specification and Design Languages (FDL'19).* Southampton, UK.

[33] Bran Selic. 1996. Real-time object-oriented modeling. *IFAC Proc. Vol.* 29, 5 (1996), 1–6. DOI:https://doi.org/10.1016/S1474-6670(17)46346-4

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

[35] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. 2018. Guidance in model-based compilations. In *Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'18), Doctoral Symposium (Electronic Communications of the EASST)*, Vol. 78.

[36] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. 2018. Towards interactive compilation models. In *Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'18)*, Lecture Notes in Computer Science, Vol. 11244. Springer, 246–260.

[37] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. 2019. Practical causality handling for synchronous languages. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'19).* IEEE.

[38] Eugene Syriani, Vasco Sousa, and Levi Lúcio. 2019. Structure and behavior preserving statecharts refinements. *Sci. Comput. Program.* 170 (2019), 45–79. DOI: https://doi.org/10.1016/j.scico.2018.10.005

[39] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. 2014. SCCharts: Sequentially constructive statecharts for safety-critical applications. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14).* ACM, 372–383.

[40] Peter Wegner. 1987. Dimensions of object-based language design. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87).* Association for Computing Machinery, New York, NY, 168–182. DOI: https://doi.org/10.1145/38765.38823

Received February 2020; revised February 2021; accepted March 2021