FROM SYNCHRONOUS SPECIFICATIONS TO ASYNCHRONOUS DISTRIBUTED IMPLEMENTATIONS

Technische Universität München
Peter Scholz

In this contribution, we sketch a design process for reactive systems, specified with the visual formalism Statecharts. In particular, we discuss how to come from a synchronous Statechart specification to an asynchronous, distributed implementation. Synchronous in this context means that these specifications are based on the assumption that all ist subcomponents interact synchronously with a global clock and in addition perfect synchronous `a la Berry.

1. Introduction

One particularly interesting application field of reactive systems are aircrafts or automobiles: reactive systems can, in the form of embedded systems, be a part of automobile electronics. In a modern upper class car, more than 50 processors or electronic control units can be integrated. These electronic systems increasingly take over tasks of the driver and assist him or her in safety critical or standard situations.

More and more functions will be implemented by software in the cars of the future. However, these systems do not act standalone, but interact to solve complex problems. Today up to 30 percent of the development costs of an upper class car are caused by electric and electronic units. Hence, every single unit must have a high performance so that the additional costs to implement it in the car are justified. Thus, there is a growing interest in industry to use not only one processor for a single task, but to appoint it to different tasks which may not be related at all. However, the specification of a task still should be implementation independent. Hence, a description technique is needed that still leaves room for various implementation

F. J. Rammig (ed.), *Distributed and Parallel Embedded Systems*

strategies, but nevertheless provides all necessary prerequisites for arbitrary implementations. Further reasons why reactive systems are implemented on distributed architectures are: the distributed implementation is imposed by the target architecture through sensor and actor locations; the distributed implementation improves the overall performance of the reactive system as the inherent concurrency in the model can be (better) exploited; through several redundant implementations a higher fault-tolerance can be achieved. This is in particular important for safety critical systems.

Altogether, the design of reactive systems and their distributed implementation is a challenging task that requires elaborated and adapted design techniques. We sketch a complete design process for the development of reactive systems with the visual formalism Statecharts. „Complete" in this context means that the designer is guided from the behavioral system specification stage to the final, distributed implementation on a processor network.

2. System Design and Implementation

To achieve this challenging task, we follow the subsequent strategy. First, we define the compositional Statechart dialect μ-Charts that is suited for both centralized and distributed implementation. For this language, a precise semantics is formalized (Scholz, 1998a) (Scholz, 1998b) . Second, we present a refinement calculus for this dialect (Scholz, 1998a). Then, we show how μ-Chart specifications can be formally verified by model checking (Philipps, 1997b) (Scholz, 1998b). Finally, we discuss how single μ-Charts specifications can be partitioned and implemented on both single processors (Philipps, 1997c) and processor networks (Scholz, 1998b). The aim of this paper is to sketch this design process in more detail, where we mainly concentrate on the presentation of partitioning.

2.1. Specification and Refinement

First of all, we develop the theory of μ-Charts. μ-Charts are a dialect of Statecharts, a visual specification language for reactive systems. In recent years, much scientific work has been invested to improve the Statecharts language. However, up to now most approaches focus more on defining new semantics for Statecharts than on providing solid specification techniques. Several informal and formal semantics for Statecharts and related dialects have been proposed.

We define the semantics $[S]_{io}$ of a μ-Chart S in terms of sets of I/O behaviors or, in other words, I/O histories (Scholz, 1998a). We illustrate that three syntactic concepts, sequential automata, signal hiding, and a composition operator including multicasting are sufficient to express also more complex Statecharts constructs; hierarchy and pure parallel composition can be defined on top of them as syntactic sugar. This strategy has several advantages. First, we restrict ourselves to the most essential language concepts and so can motivate that Statecharts are not as complex as assumed in the hitherto existing literature. Second, we get a semantics that leads to simple proofs

when reasoning about the semantics itself. Third, treating hierarchical decomposition as parallel composition and communication is an elegant way to establish distributed implementation of charts in different hierarchical levels.

Our μ-Charts formalism is considered to be a specification technique rather than a (synchronous) programming language like Argos, ESTEREL, LUSTRE, SIGNAL. Therefore, we show how to use μ-Charts in an incremental development process. We demonstrate what it means to develop a system step-by-step. We present a refinement calculus with rules that are easy to understand, but at the same time describe formal design steps towards the final system. One of our goals is to show that μ-Charts are more than a simple two-dimensional programming language. A design methodology is needed, supported by a set of purely syntactic refinement rules that tell the user how to transform an abstract, under-specified system description into a more concrete one. The essential rules we presented in (Scholz, 1998a) are thought to be applicable not only for μ-Charts, but also for any other version of Statecharts. On the semantic level, under-specification is reflected by non-determinism.

2.2. μ-Charts

Statecharts are a graphical description technique for the state-based, behavioral specification of control-oriented systems. In this section, we introduce a Statecharts-like language, termed μ-Charts, which we have developed as basis for design and distributed implementation of reactive systems.

The main concept of Statecharts are sequential automata or, in other words, Mealy machines A. These automata consist of states and transitions. An automaton's state denotes a section of a complex state of a reactive system. Transitions connect those states that describe consecutive system states. A transition can be labeled with a pair, consisting of the condition that must be fulfilled in order to establish the transition and an action that specifies more detailed system behavior when taking the transition. Most graphical notions follow the convention that this pair is separated by a slash, *condition / action*, and we adopt this notation for μ-Charts, too.

To enable description of practically relevant systems, these automata can be composed to larger specifications using two principal structuring mechanisms: parallel composition and hierarchical decomposition.

For parallel composition, the basic assumption is that sub-charts S_1 and S_2 composed by $S_1 \bullet S_2$ in parallel proceed in lock-step with respect to a common clock. Without any further assumptions, these automata do not interact at all. However, if specified by the user, they also can interchange messages L (L is the set of those signals that can be sent and received by S_1 and S_2) in order to influence the behavior of each other: $S_1 \bullet L \bullet S_2$. Note that $S_1 \bullet S_2$ is just a special case for $S_1 \bullet 0 \bullet S_2$ and therefore defined as syntactic abbreviation.

Reactive systems possibly have complex system states. Hence, single automata states may not be appropriate for a detailed description of these systems, and more elaborated techniques are needed. Statecharts enable to further structure single states of sequential automata by hierarchical decomposition. The Statechart that describes the systems behavior in a specific state in more detail is simply (graphically) inserted

into this state. The behavior of an hierarchically decomposed Statechart is comparable with the one of procedure calls in an imperative programming language. An automaton of an higher level of hierarchy „calls“ sub-routines, that is, Statecharts of a lower level, whenever a decomposed state is entered.

To express hierarchical decomposition, which plays a key role in the concept of Statecharts, we do not need a principally new syntactic construct, but derive hierarchy from the composition operator. This facilitates the definition of both formal semantics and refinement calculus. Furthermore, to express hierarchy by parallel composition and multicasting is a convenient way to provide a basis to implement charts of different levels of hierarchy on different processors. With Statecharts semantics that treat hierarchy on the semantic level this could not be achieved that elegantly. In (Scholz, 1998a), it is explained in detail how hierarchical decomposition can be defined by composition and communication.

2.3. Verification

In former publications (Philipps, 1997a), we also demonstrated how to use the semantic model we introduced as starting point for an efficient formal verification, based on symbolic model checking techniques. We gave a scheme, demonstrated by an example, that translates μ-Charts specifications into μ-calculus (Biere, 1997) formulae and the SMV (McMillan, 1993) input language.

2.4. Implementation

As a further result of our work on symbolic verification of μ-Charts, we showed how a μ-Chart can be implemented as a finite state machine, using a register and a combinational logic block that represents the transition relation of the system (Philipps, 1997c).

However, we do not only provide a strategy for centralized implementation, but in addition discuss problems that occur and restrictions that have to be fulfilled when a μ-Chart specification will be realized with more than one processor. The architecture template we focus on is a realistic architecture that is widespread in automobile industry: it consists of a number of electronic control units that get inputs from sensors, deliver outputs to actors, and are connected via one or more field busses.

2.5. Partitioning

Before a single μ-Chart specification can be implemented on this target architecture, however, some deliberation is necessary. We discuss which problems occur when a specification is partitioned in order to implement it on a processor network.

Apart from the aforementioned reasons, we had to develop our own dialect of Statecharts to bridge the gap between an abstract, formal system description and a

distributed implementation. The μ-Chart language fulfills several technical requirements that are a prerequisite for distributed implementations.

It only consists of three syntactic constructs: sequential automata, signal hiding, and a composition operator that combines parallel composition and multicasting. Hierarchical decomposition is defined as syntactic abbreviation: it is achieved by parallel composition and message passing. Thus, we can solve the question how to implement charts of different levels of hierarchy on different processors. With Statechart semantics that treat hierarchy on the semantic level this could not be achieved that elegantly. A syntactically rich language, in which every single construct has its own semantics would neither be suited as starting point for the definition of refinement techniques nor for partitioning nor for distributed implementation.

A further prerequisite for distributed implementation is compositionality of the underlying description technique. Our μ-Chart semantics is compositional and therefore builds a solid basis for partitioning.

Since their syntax is defined incrementally with sequential automata as basic construct, μ-Charts straightforwardly lend themselves to partitioning specifications on the granularity level of automata. A more fine-grained granularity would yield a far too complex partitioning strategy and a more coarse-grained granularity possibly would yield inefficient partitionings.

When partitioning a specification in μ-Charts, care has to be taken that communication between the different parts is deadlock-free or, in other words, does not contain any causality errors. The fixed point semantics we define for the deterministic μ-Charts version exactly reflects the signal causality relationships within one instant and therefore also the communication of distributed μ-Charts components in detail. As a consequence, this semantics serves as mathematical basis to value the partitioning and to verify that no communication deadlocks arise.

As the overall result, this paper offers a design process for a Statecharts dialect that spans the gap between visual formalization of reactive systems up to their distributed implementation on a processor network in an embedded system. Apart from the very early development phase requirements engineering, all important stages of system development (description, refinement, formal verification, partitioning, and implementation) are covered. In each design step, we use equivalent formal semantics for μ-Charts. As a consequence, we do not loose any information between the different stages. Thus, the designer can be sure that the system he or she gets in the final product is semantically equivalent to the one that has been abstractly specified, refined, and formally verified. Hence, we follow the principle of „what you specify and verify is what you implement“ and all concepts presented here harmonize with each other.

2.6. Entire Design Process

Our design process for reactive systems starts with the state-based *system description* by the Statecharts-like specification language μ-Charts. As a user might not have a precise imagination how the system under development shall look like in detail in this phase, the system yet might be *under-specified.* Roughly speaking, this means that design alternatives for certain system parts or, more precisely, certain sequential automata yet have been left open by him or her. In the context of this paper, under-specification is a purely syntactic notion that only refers to sequential automata. An automaton can be under-specified because of the following three reasons.

First, the user may have specified more than one initial state for the automaton. Second, the designer may have specified more than one transition for at least one input event in one or more states. Last but not least, no transition might be defined for at least one input event in one or more states. The last type of under-specification is termed *non-responsiveness.* If for every state and every input event of an automaton at least one transition has been specified, we call the automaton *responsive* because it can fire a transition in every state on every input event. This is not the case for non-responsive automata. In the first two cases of under-specification, we say that the system under development is *(syntactically) non-deterministic.* An automaton where exactly one transition is defined for each possible input event in each state and that only has one single initial state is both *deterministic* and *responsive.* We call such an automaton also *completely specified.* If all automata of a μ-Chart specification are completely specified, we also call the overall μ-Chart itself completely specified and under-specified otherwise. If all automata are responsive, we also term the μ-Chart itself responsive and non-responsive otherwise.

Here, we would like to add a warning: apart from this purely syntactic notion of non-determinism, there also exists a semantic notion of non-determinism. A μ-Chart is *(semantically) non-deterministic* if and only if its semantics contains more than one input/output history (see (Scholz, 1998a)). For sequential automata, under-specification is isomorphic to semantic non-determinism. However, for the composition operator of μ-Charts this is in general not true. Later on, we will see that syntactic non-determinism also is semantic non-determinism, but that the opposite is not true. With μ-Charts, semantic non-determinism cannot only be caused by syntactic non-determinism or, more generally, under-specification, but also by composition. This phenomenon is common to most synchronous languages. While syntactic non-determinism is intended by the user, semantic non-determinism that is caused by composition is not. Detecting semantic non-determinism caused by composition requires subtle analysis techniques which have been discussed in detail for centralized implementations in the literature (cf. ESTEREL). In this paper, we will present an approach for distributed implementation.

We recognize that non-responsive automata do not have a defined semantics since the designer did not define any behavior. In order to ease the mathematical formulation of the behavior, that is, the semantics, and the mathematical reasoning with it, we provide a technique to unify both syntactic notions, non-determinism and non-responsiveness, on the semantic level. The technique we use is *chaos completion* (Scholz, 1998b). It allows to interpret non-responsiveness as non-determinism, too. Whenever a sequential automaton of a μ-Charts specification is non-responsive in a state, we assume that this automaton can perform every possible reaction in this state,

that is, outputs an arbitrary event that is included in its output interface and has an arbitrary successor configuration. The latter means that the subsequent control and data states can have an arbitrary value within their admissible ranges.

Instead of no system reaction we so get all possible system reactions. Therefore, we say that this part of the system behaves *chaotically*. The mathematical result of the discussion above is a stream semantics for μ-Charts with chaos completion. This stream semantics is tailored for both the mathematical definition of the behavior of possibly non-deterministic and non-responsive μ-Chart specifications and the formal reasoning about how to get implementable specifications: since reactive systems must behave reliably and completely predictably in all situations, we want our final systems to behave (semantically) deterministic.

As a consequence, in the course of system development, under-specification has to be resolved in order to get an implementable system. Hence, in the next design phase, the *refinement* phase, we provide the system developer with easy-to-apply refinement techniques. These are based on a set of syntactic refinement rules. Each rule supports the designer with context restrictions that must be fulfilled in order to get correct refinement steps. All rules are combined to a *refinement calculus* (Scholz, 1998a) that guarantees the correct step-wise concretization of a system under development towards a final deterministic and responsive description that fits for implementation. We would like to mention that through refinement also those specifications that include (unintended) semantic non-determinism that has been introduced through composition may be transformed into deterministic specifications.

As soon as the system engineer comes up with a responsive specification, which yet may be non-deterministic, she or he can begin to formally verify certain properties of the system by *model checking* (Phillips, 1997b) (Scholz, 1998b). Note that we do not support the model checking of non-responsive μ-Charts. This is because of methodological and technical reasons. If no behavior is specified, any system reaction is possible. Hence, the system does not have any longer a „reasonable" behavior, for which interesting properties could be verified. Moreover, since our semantic view of non-responsiveness is chaos, we would have to explicitly encode chaotic behavior with the input language of a model checker, too. If we skipped these explicit encodings, our model for formal verification, a Kripke structure, possibly could perform deadlocks in those configurations where the μ-Charts is non-responsive. However, the explicit encoding of chaos is impracticable. As a consequence, we decided to support model checking of responsive μ-Charts only, which is not a strong restriction in practice. Notice that in contrast to the explicit encoding for model checking, we can formulate chaos implicitly with the theoretical semantics.

Model checking must not necessarily be performed in a separate phase of development shortly before the implementation. Rather, system properties of different levels of abstraction can be verified at different phases of development. These properties can be divided into liveness and safety properties. Informally, the first guarantees that eventually something good will happen and the latter that nothing bad will happen. Once having verified a certain safety property, the application of correct refinement rules guarantees that this property still holds after the application of this rule. While safety properties are not affected by refinement steps, liveness properties are. This is explained by the fact that non-determinism is restricted through

refinement. As a consequence, behavioral alternatives that have been included on an earlier design stage possibly are excluded by subsequent refinement steps.

When we finally arrive at a complete specification for which all required properties have been formally verified, we can implement it either on a centralized or on a distributed target architecture. While the first is relatively unproblematic, this is not the case for the latter. Locations for every single sequential automaton have to be defined. We call this process, which can be optimized by tool support, *allocation*. However, before a complete specification can be implemented on a distributed target architecture, a number of considerations is needed. First, this specification has to be partitioned. However, not every partition is feasible. In (Scholz, 1998b) and Section 2.2, we discuss context restrictions and provide the system engineer with guidelines how to partition a μ-Chart specification.

In order to reason about partitioning and distributed implementation, we consult the formal semantics of μ-Charts again. However, the mathematical discussion of multicast communication in the case of distributed implementation requires a slightly different semantic view: in each instant, distributed components send and receive signals according to a chain reaction until the overall system reaches a stable configuration. This stabilization process can be mathematically understood as fixed point computation. Hence, we provide a tailored semantics for deterministic and responsive μ-Chart specification to do so. For deterministic and responsive specifications this semantics is just a concretization of the original stream semantics.

3. Partitioning

In this section, we discuss the problems that potentially may occur when one tries to partition a design with Statecharts and to implement the different parts on a processor network. Problems of this type are the lack of a global clock for a loosely coupled network, communication deadlocks, and the scanning for signal absences in synchronous languages. Note that these aspects are not problematic in case of centralized implementations, but are common to most synchronous languages.

Problems

Partitioning a specification written in a synchronous language, we are faced with a number of interesting problems that are common to most synchronous languages, such as ESTEREL, LUSTRE, SIGNAL, Argos, and μ-Charts. In (Scholz, 1998b), we discussed these problems in detail by means of μ-Charts and presented strategies to remedy them. These strategies provide partly automated solutions and partly syntactic restrictions to the design process of reactive systems with synchronous languages. We begin with a classifications of the envisaged problems and then sketch appropriate solutions.

First, synchronous languages are based on a perfect synchronous time model (Berry, 1998); in particular, the assumption is that all components of a specification interact in lock-step. While this assumption can be fulfilled when the specification is implemented on centralized and tightly coupled target architectures, respectively, this

is not the case for loosely coupled systems. Distributed processors in embedded systems, as they occur in automobile industry, are mostly loosely coupled: they communicate via one or several field busses. For these systems, we cannot longer assume that all distributed components are triggered by one single global system clock. Hence, it is difficult to guarantee that also distributed implementations of synchronous specifications interact in lock-step. In particular, care has to be taken that all distributed components start to react in the same instant. Since external input signals are responsible to start reaction in each step, we have to examine these signals and their mutual dependence in more detail.

Second, while external signals are produced by the system environment, internal signals are sent from other system components. The classification into external and internal signals plays an important role in our analysis. If components written in a synchronous language are partitioned and implemented on a distributed architecture, the logical communication of the specification becomes a physical communication. Now, it is important to guarantee that this communication, which possibly consists of a number of sending and receiving actions and in which a number of components can be involved, does not have deadlocks.

The third problem is common to external and internal signals. As a global clock is not available in loosely coupled systems, it is not possible for a distributed reactive system to react on the absolute absence of a signal. The problem here is to decide when to react on the absence of a signal; it is difficult to argue about the relative presence or absence of external signals. As a consequence, we require for distributed implementation that each component can only scan single external input signals in each instant, but not groups of them.

Solutions

To overcome the problems with *external input signals* of distributed implementations, we have proposed two different strategies. We can either restrict the feasible trigger conditions of sequential automata to conditions that only can react on single input signals or we can make assumptions to the system environment by using an rely/guarantee-like specification style. In the former case, give syntactic restrictions that help the designer to specify sequential automata in a way that they always react on just one single external input signal. However, for composed charts we cannot give easy-to-understand syntactic guidelines. Here, tool support is necessary. In (Scholz, 1998b), we discuss how this in principle has to look like. Since for some reactive systems it may be too restrictive to require that its distributed implementation merely reacts on the presence of single signals, but also on the relative presence or absence of more than one, we sketch how this restriction can be weakened using additional information about the external signals of the environment. Of course, in principle, a centralized implementation has to overcome the same problem because its input interface also has to collect signals to events, that is, sets of signals. However, this can be done much easier by a clock signal in the case of centralized implementation. Hence, the problem with external input signals can be avoided for centralized implementations. Using the same clock for all processors of a loosely coupled distributed target architecture is not possible due to technical reasons.

The same argument with the global clock also applies to the problem of dealing with *signal absences* using only one single clock in a centralized implementation enables the scanning of signal absences.

However, the situation is somewhat different in case of the problems *internal input signals* cause in distributed implementations. While for centralized implementations it is sufficient to know that a specification or, more precisely, its semantics, is continuous, we had to require additionally for distributed implementations that their signal graphs, that is, visualizations of the signal dependencies, do not contain any harmful cycle in any reachable configuration. This is necessary to avoid communication deadlocks. In (Scholz, 1998b), we provided an algorithm to detect deadlocks that is similar to the approach for causality analysis suggested by ESTEREL. For centralized implementations, this kind of causality analysis is not needed. Here, already monotonicity alone is adequate, because monotonic charts always have a least fixed point. In order to generate a centralized implementation this is a sufficient property. Since signal graphs can only be constructed for monotonic charts, the existence of a signal graph without harmful cycles also guarantees the chart's monotonicity.

Last but not least, we give syntactic guidelines that support the specifier in writing specifications that do not cause problems in case of distributed implemented due to scanning of absolute absence of external or internal input signals.

In (Scholz, 1998b), we have discussed these three problems and their solutions in more detail. Here, we merely could give a short overview of our solutions due to space limitations.

4. Conclusion

This work was driven by the idea of supporting a design method for Statecharts that covers the design phases description, refinement, verification, and generation of centralized and distributed code, independently of whether the code is implemented in hardware or software. As a first step towards this goal we developed the visual formalism μ-Charts, a dialect of Harel's Statecharts. In contrast to Statecharts and many related approaches, μ-Charts overcome many semantic problems inherent in the former notation. The core language of μ-Charts consists of only three constructs: sequential automata, signal hiding, and a composition operator.

Starting with a lean core language and then extending it by means of abbreviation mechanisms has several advantages. First, it eases the task of obtaining specifications that can be partitioned and implemented on a distributed processor network. Second, it eases reasoning about the semantics itself. Finally, the task of partitioning a monolithic specification is simplified.

5. Acknowledgement

I thank Martin Rappl for carefully reading a draft version of this paper.

References

Berry, G. (1998). *The Foundations of Esterel*, In G. Plotkin, C. Stirling, and M. Tofte, editors, Proof, Language and Interaction: Essays in Honour of Robin Milner, MIT Press.

Biere, A. (1997). *μ-cke - Efficient μ-Calculus Model Checking*. Number 1254 in Lecture Notes in Computer Science, pages 468-471, Springer Verlag.

McMillan, K.L. (1993). *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University.

Philipps, J., Scholz, P. (1997a). *Compositional Specification of Embedded Systems with Statecharts*. In TAPSOFT'97: Theory and Practice of Software Development, volume 1214 of Lecture Notes in Computer Science, Springer Verlag.

Philipps, J., Scholz, P. (1997b). *Formal Verification of Statecharts with Instantaneous Chain Reactions*. In TACAS'97: Tools and Algorithms for the Construction and Analysis of Systems, volume 1217 of Lecture Notes in Computer Science, Springer Verlag.

Philipps, J., Scholz P. (1997c). *System-Level Hardware Design with μ-Charts*. In CHDL'97: Hardware Description Languages and their Applications, Toledo.

Scholz, P. (1998a). *A Refinement Calculus for Statecharts*. In Proceedings of the „ETAPS/FASE'98, Lisbon (Portugal), March 30 - April 03, 1998", volume 1382 of Lecture Notes in Computer Science, Springer Verlag.

Scholz, P. (1998b). *Design of Reactive Systems and their Distributed Implementation with Statecharts*. PhD thesis, Technische Universität München, TUM-I9821.