A Refinement Calculus for Statecharts *

Peter Scholz
scholzp@forsoft.de

Technische Universität München, Institut für Informatik
D-80290 München, Germany

Abstract. We present a Statecharts dialect with only three syntactic constructs and a semantics that is not restricted to describe reactive systems on an implementation level but allows to model them on an abstract, more specification oriented stage, where design alternatives are still left open. We give a refinement calculus with rules that tell the designer how to come from the abstract specification to the implementation such that the system under development only becomes more concrete but not more abstract; under-specification is eliminated by adding more information. The result of a design process that follows these rules is an implementation that satisfies its specification by construction.

1 Introduction

Statecharts [5] are used in industry to develop reactive systems. A typical application area is rapid prototyping of embedded systems as they occur in avionics and automobile industry. Among other things, their success comes from two facts. First, it is an easy to learn language for design specialists who have more often a degree in electrical or electronic engineering than a solid background in computer science. Those engineers have a considerably better intuition of the meaning of automata than of algebraic specification techniques, for instance. Second, Statecharts are available as description technique in commercial products, like Statemate. Therefore, such specifications are really sturdy for engineers.

In the past years, much scientific work has been invested to improve the Statecharts language. However, up to now most approaches focus more on the implementation aspects of Statecharts than on specification techniques. Several formal semantics for Statecharts and related dialects have been proposed (see [13] for a good but no longer complete overview). Among them some approaches like Argos [7, 8] can be found that are closely related to the reactive programming language Esterel [2, 3].

Our μ-Charts exclude a number of syntactical concepts of Statecharts (as presented in [5]) that lead to semantical problems, such as inter-level transitions,

* This work is partially sponsored by the Bavarian Research Foundation (Bayerische Forschungsstifung) as part of the compound project "ForSoft".

priority of transitions w.r.t. state hierarchy, multiple source and target of transitions and so on. When designing our dialect μ-Charts we were inspired by Argos but also tried to modify some basic concepts as discussed in [9]. Our μ-Charts formalism is considered to be a specification mechanism rather than a programming language like Argos, Esterel, Lustre, or Signal. In these synchronous programming languages *unintended* non-determinism that is obtained by composition is avoided by static analysis.

Besides this unintended non-determinism that stems from composition there is also *intended* non-determinism to express underspecification of components. Intended non-determinism is volitional by the user and reflects that design decisions for a component are still left open at the current level of development.

In [9] we have defined the semantics of μ-Charts in terms of sets of I/O-behaviors or, in other words, I/O-histories. The μ-Charts semantics presented here differs in some points with the semantics published in [9]. These modifications have been necessary for a smooth integration of refinement.

In this contribution, we further improve our language concepts. We illustrate that three principal syntactical concepts, sequential automata, hiding, and a composition operator including multicasting, are enough to express more complex Statecharts' constructs; hierarchy and pure parallel composition can be defined as syntactic sugar. This strategy has two main advantages: First, we reduce ourselves to the most essential language concepts and so can motivate that Statecharts are not that complicated as assumed in the hitherto existing literature. Second, we get an easy semantics for the proofs of the refinement rules' soundness.

Moreover, we show how to use this specification formalism in the development process. We demonstrate what it means to incrementally 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. Though this paper is rather theoretically written to motivate that all concepts are sound, also more practical oriented readers should gain from reading this article: For those readers it should be enough to understand which syntactical side conditions have to be fulfilled to make a certain refinement step.

Our goal is to underline that Statecharts are more than a simple twodimensional programming language. What is needed is a design methodology, supported by a set of refinement rules that tell the user how to come from an abstract system description to a more concrete one. In principle, the essential rules we present (for hierarchical decomposition and parallel composition, for instance) are thought to be applicable not only for μ-Charts but also for any other version of Statecharts.

This paper is organized as follows: Section 2 contains the running example, which is used to underline our refinement technique. In Section 3.1 and 3.2 we explain syntax and semantics of the μ-Charts language. The refinement rules are discussed in Section 4. We finally conclude this paper with Section 5.

2 Running Example

As running example we take a simplified specification of a realistic central locking system for two-door cars as already used in [10]. We are aware that this example is much smaller than industrially relevant examples. However, it should be large enough to illustrate all essential points of this contribution. Due to space limitations we are not able to print a larger example.

The system architecture is pictured in Figure 1 and the corresponding μ-Chart in Figure 2. Our graphical syntax for μ-Charts follows the convention that ellipses denote basic states of sequential automata while boxes denote states that are decomposed by other μ-Charts. Later on, we will show that decomposition is in fact only syntactic sugar. Double frames denote default states. Notice that, as we deal with underspecification, more than one default state is possible as demonstrated in the example.

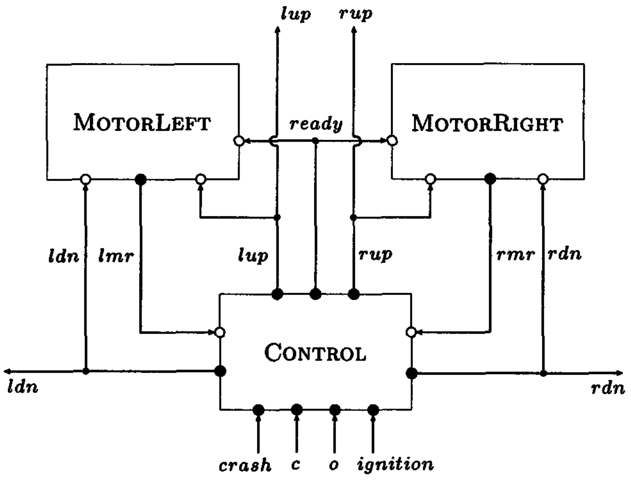

Figure 1. Central locking system — Architecture

Our central locking system consists essentially of three main parts (see Figure 1): the CONTROL and the two door motors. These parts react independently. The default configuration of the system is that all doors are unlocked (UNLD) or locked (LOCKED) and both motors are OFF. Depending on the system's configuration, the driver can (un-)lock the car either from outside by turning the key or from inside by pressing a button. Both actions generate the external signal (*o*) *c*. The CONTROL generates the internal signals *ldn* and *rdn* and enters its locking state LOCKG, which is decomposed by the automaton in Figure 6.

Whenever the crash signal occurs, the CONTROL changes from the NORMAL mode in the CRASH mode and generates the signals *lup* and *rup* and the doors

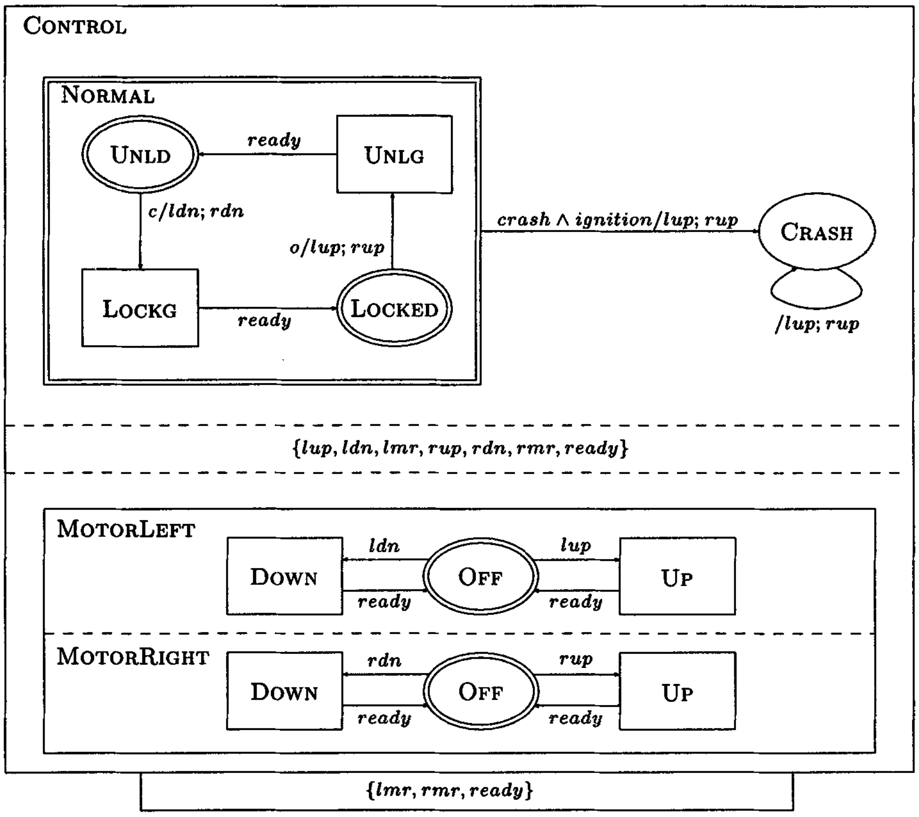

Figure 2. Central locking system — Behavior

will open. This property has been proven applying model checking techniques in [10]. A more detailed informal description of the case study also can be looked up there.

3 The μ-Charts Language

3.1 Syntax

In this section we briefly introduce the essential concepts of our Statecharts dialect. We assume the reader to be familiar with the basic ideas of Statecharts and refer to [5, 6] for a more detailed introduction.

In this paper, all elements in the set S of μ-Charts can be built from only three syntactical constructs: non-deterministic sequential automata, hiding, and parallel composition including communication between parallel composed charts.

Sequential Automata. In the definition of sequential automata, we use the following syntactical, pairwise disjoint sets: *Ident* is a set of identifiers, *Signals* a set of signal names, *States* a set of state names, and V a set of variable names. The construct $(N, I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta)$, in the sequel abbreviated to A, is an element of $\mathcal{S}$ iff the following constraints hold:

1. $N \in \mathit{Ident}$ is the unique identifier of the automaton.
2. $I \subseteq \mathit{Signals}$ is the input interface.
3. $O \subseteq \mathit{Signals}$ is the output interface. We assume I and O to be disjoint.
4. $\Sigma \in \wp(\mathit{States})$ is a nonempty finite set of all control states of the automaton.
5. $\Sigma_0 \subseteq \Sigma$ represents the set of initial states.
6. V_l is the set of local (integer) variables of the automaton.
7. For each initial state $\sigma_0 \in \Sigma_0$ the function $\varphi_0(\sigma_0) \in V_l \rightarrow \mathbb{Z}$ initializes the local variables. We abbreviate $V_l \rightarrow \mathbb{Z}$ to $\mathcal{E}(A)$.
8. $\delta : \Sigma \rightarrow \wp(\mathit{Bexp}(I + V_l) \times \mathit{Com} \times \Sigma)$ is the finite state transition relation that takes a state and yields a set of triples, where each triple consists of a Boolean expression over I and V_l as transition predicate (guard, pre-condition, trigger) paired with a command $\mathit{com} \in \mathit{Com}$ and the successor (control-)state.

In this context, arithmetic expressions $a \in \mathit{Aexp}$, Boolean expressions $b \in \mathit{Bexp}$, and commands $c \in \mathit{Com}$ have the form:

$$\begin{aligned} a &::= n \mid Y \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \\ b &::= \mathsf{true} \mid \mathsf{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid s_i \mid \neg b \mid b_1 \wedge b_2 \\ c &::= \mathsf{skip} \mid Y := a \mid s_o \mid c_1 ; c_2 \end{aligned}$$

In the syntax of transitions we have followed the convention that $n \in \mathit{Int}$, $Y \in V_l$, $s_i \in I$, and $s_o \in O$. Note that to permit not only integer variables in *Int* but arbitrary types is a straightforward extension but is not relevant in the context of this paper and therefore was omitted. The meaning of these expressions and commands is straightforward. In contrast to [6], we use the semi-colon as sequential and not as parallel composition and so avoid racing conditions.

Composition. Suppose that $S_1, S_2 \in \mathcal{S}$ are arbitrary μ-Charts and L is the set of signals that can be possibly transmitted, then the composition $S_1 \lhd L \rhd S_2$ is also in $\mathcal{S}$. Defining the semantics of this operator, we will see that instantaneous communication [2, 3, 7] is achieved by signal feedback (see also Figure 3). Graphically, this construction is denoted as signal set between the dashed lines that separate S_1 and S_2. Though one is totally free in the choice of L, it should be a subset of $(\mathit{In}(S_1) \cap \mathit{Out}(S_2)) \cup (\mathit{In}(S_2) \cap \mathit{Out}(S_1))$ to get meaningful specifications. Here, $\mathit{In}(S_i)$ and $\mathit{Out}(S_i)$ denote the input and output interfaces of S_i, respectively.

Elements in $\mathit{In}(S)$ and $\mathit{Out}(S)$ are called *input* and *output signals*, respectively. If we do not care about the flow direction, we only say *signal*. Each element x

in $\mathcal{I}(S) =_{df} \wp(In(S))$, and $\mathcal{O}(S) =_{df} \wp(Out(S))$ is called an *input* and *output event*, respectively. If we abstract from input or output, we simply speak of *events*. For each signal s an we say s is *present* in event x iff $s \in x$. Otherwise, we say that it is *absent* in x.

Hiding. Output signals that are sent using the ternary operator $. \lhd . \rhd .$ are still visible by the environment of the chart S. If the signal set K shall be hidden for the part of the specification not belonging to S, we use the hiding operator $[S]_K$. The construct $[S_1 \lhd L \rhd S_2]_L$, for instance, hides *all* output signals that are fed back. Likewise for communication, there is also a graphical counterpart for hiding; it is a box, attached to the bottom of S, which contains the signals K that are hidden.

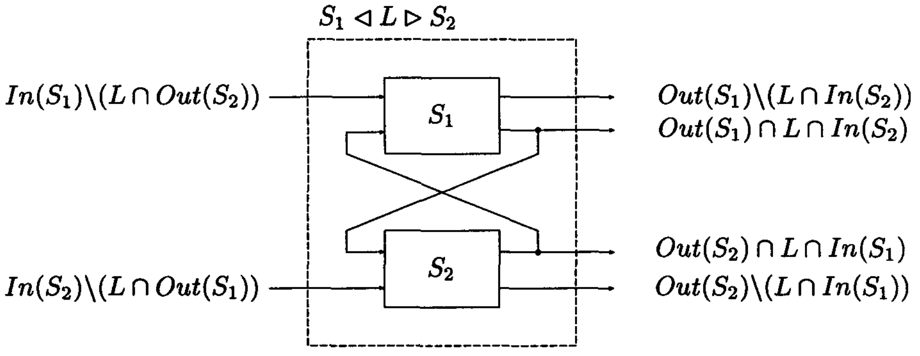
$S_1 \lhd L \rhd S_2$
$In(S_1)\backslash(L \cap Out(S_2))$
$In(S_2)\backslash(L \cap Out(S_1))$
S_1
S_2
$Out(S_1)\backslash(L \cap In(S_2))$
$Out(S_1) \cap L \cap In(S_2)$
$Out(S_2) \cap L \cap In(S_1)$
$Out(S_2)\backslash(L \cap In(S_1))$

Figure 3. Composition

Hierarchy. To express hierarchical composition, which plays a key role in the concept of Statecharts, we do not need an explicit syntactical construct but derive hierarchy from the above composition operator. This facilitates the definition of both formal semantics and refinement calculus. How this can be achieved will be demonstrated in the sequel.

A hierarchical decomposition A decby (Σ_d, ϱ), where $A = (N, I, O, \Sigma, \Sigma_0, V_l, \varphi_0, \delta)$ can be expressed by composition. Here, the total function $\varrho : \Sigma_d \to S$ defines for each state in $\Sigma_d \subseteq \Sigma$ the sub-chart by which it is decomposed. Thus, hierarchy can be considered as abbreviation mechanism. We can translate hierarchical specifications in flat ones applying the following algorithm (hereby we call A the *master* and $\varrho(\sigma)$ with $\sigma \in \Sigma_d$ the *slave*). The algorithm works differently for weak and strong preemption [1]:

1. Master A and all slaves are composed in parallel.

2. Modification of the master: The fresh signals $go(\sigma)$ for all decomposed states $\sigma \in \Sigma_d$ are added to the output interface of A. In case of weak preemption, for every $\sigma \in \Sigma_d$ the command *com* of every outgoing edge is replaced by $com; go(\sigma)$. Here, $go(\sigma)$ is a signal which indicates that the slave attached to σ is currently active and is allowed to fire its transitions. This modification is omitted for other edges than self loops when strong preemption is desired. Let t_σ be the disjunction of all trigger conditions on all outgoing edges (inclusive self loops, if they exist) of σ. Then, for every kind of preemption, additionally every state σ is enriched by a self loop with trigger condition $\neg t_\sigma$ and command $go(\sigma)$.
3. Modification of the slave(s): For every $\sigma \in \Sigma_d$ and every sequential automaton in $\varrho(\sigma)$ to every input interface the signal $go(\sigma)$ has to be added. Furthermore, every trigger condition t on every edge has to be substituted by $t \wedge go(\sigma)$ in order to guarantee that the now parallel composed slave only reacts iff it is allowed to. If σ is a non-history [5] decomposed state, additional transitions from every state but the default state of every sequential automaton in $\varrho(\sigma)$ with label $\neg go(\sigma)/\varphi(\sigma)$ have to be introduced. This is necessary to initialize the slave whenever the master is left. Otherwise, all slaves would stay in their current states when the master changes its current step; this, however, is only wanted for history decomposed states [5].
4. In order to enable the communication between master and slave, all above introduced *go* signals have to be fed back and hidden with respect to the parallel composition of the master and all slaves. Feedback applies to all signals in $In(A) \cap Out(\varrho(\sigma))$, too, to enable communication from slave to master.

Figure 4 shows an example. It is the CONTROL part of the locking system. As once having entered the CRASH mode, the system resides in this state forever. Thus, it makes no difference whether NORMAL is history or non-history decomposed. Figure 4 shows weak preemption; if we omitted the statement $go(\text{NORMAL})$ on the transition between NORMAL and CRASH, we would model strong preemption. In case of nested hierarchy, this algorithm has to be recursively applied.

3.2 Semantics

Steps and System Reactions. Like other Statecharts dialects, μ-Charts are a synchronous language based on a discrete, clock-synchronous time model. It follows the principles of the perfect synchrony hypothesis [3] and uses, similar to [8], instantaneous feedback as semantical model for communication. A *system reaction* of a μ-Charts consists of a sequence of *steps (instants)*. At each step, the system receives a set of signals from the environment. Upon reception of this input set, the system produces a set of output signals, modifies local variables, and changes its control state. The output signals are assumed to be generated in the same instant as the input signals are received. A signal is said to be *present*

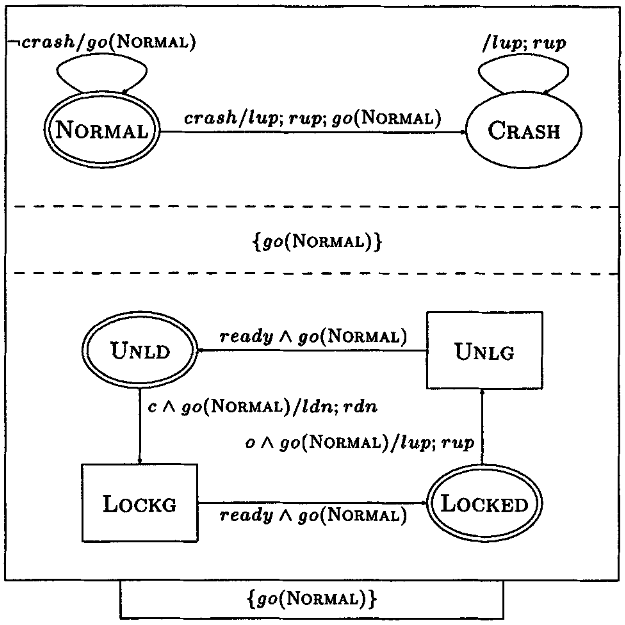

Figure 4. Unpacked hierarchy

in a given instant, if it is either input from the environment or generated by the system. Otherwise, it is said to be *absent.*

Reactive Behavior. Reactive systems have to interact continuously with the environment. Hence, their complete input/output behavior can be described using communication histories. We model the communication history of μ-Charts by streams carrying sets of signals. Mathematically, we describe the behavior of μ-Charts by relations over streams. Thus, we briefly discuss the notion of streams. For a detailed description we refer, for example, to [4].

Given a set X of signals, a stream over X, denoted by X^∞, is an infinite sequence of elements from X. Our notation for the concatenation operator is &. Given an element x of type X and a stream s over X, the term $x\&s$ denotes the stream that starts with the element x followed by the stream s.

For a chart $S \in \mathcal{S}$ we denote the non-deterministic I/O-behavior as relation $[\![S]\!]_{io} \in \wp(\mathcal{I}(S)^\infty \times \mathcal{O}(S)^\infty)$. This pure I/O-semantics is defined by using the auxiliary relation $[\![S]\!] \in \wp(\mathcal{C}(S) \times \mathcal{I}(S)^\infty \times \mathcal{O}(S)^\infty)$:

$$[\![S]\!]_{io} =_{df} \{(i,o) \mid \exists c.c \in \mathit{Init}(S) \wedge (c,i,o) \in [\![S]\!]\}$$

where $\mathit{Init}((N,I,O,\Sigma,\Sigma_0,V_l,\varphi_0,\delta)) =_{df} \{(\sigma_0,\varphi(\sigma_0)) \mid \sigma_0 \in \Sigma_0\}$ and $\mathit{Init}(S_1 \lhd L \rhd S_2) =_{df} \mathit{Init}(S_1) \times \mathit{Init}(S_2)$ denote the initial configurations. A *configuration* of chart S is an element in $\mathcal{C}(S)$, which is inductively defined by:

$$\begin{aligned}\mathcal{C}((N,I,O,\Sigma,\Sigma_0,V_l,\varphi_0,\delta)) &=_{df} \Sigma \times (V_l \to \mathbb{Z})\\ \mathcal{C}(S_1 \lhd L \rhd S_2) &=_{df} \mathcal{C}(S_1) \times \mathcal{C}(S_2)\end{aligned}$$

Instead of the explicit tuple we often simply write c to denote an arbitrary configuration. The auxiliary semantics of a sequential automaton A is now defined as the greatest solution of the following recursive equation $(*)$:

$$[\![A]\!] = \{(c, x\&i, y\&o) \mid \exists c'.((c',y) \in [\![\delta]\!](c,x) \wedge (c',i,o) \in [\![A]\!]) \vee [\![\delta]\!](c,x) = \emptyset\}$$

Informally, $[\![A]\!]$ is the set of all those tuples $(c, x\&i, y\&o)$ such that one of the following two cases is true. Either A generates the output event y and changes its current configuration from c to c' while reacting on input event x and then behaves similarly in the new configuration c' 'eating' the rest of the input event stream or the reaction is (yet) underspecified: In this case, the predicate $[\![\delta]\!](c,x) = \emptyset$ is a characterization for the chaotic behavior of A, as the choice of i, y, and o is not restricted at all. Here, $[\![\delta]\!]$ is defined from the transition relation δ as follows[1]: For all $c = (\sigma,\varepsilon) \in \mathcal{C}(A)$ and $x \in \mathcal{I}(A)$:

$$\begin{aligned}[\![\delta]\!]((\sigma,\varepsilon),x) =_{df} \; &\{((\sigma',\varepsilon'),y) \in \Sigma \times \mathcal{E}(A) \times \mathcal{O}(A) \mid\\ &\exists t, com.(t, com, \sigma') \in \delta(\sigma) \wedge (\varepsilon,x) \models t \wedge (\varepsilon',y) = \mathcal{R}[\![com]\!]\varepsilon\}\end{aligned}$$

$[\![\delta]\!]((\sigma,\varepsilon),x)$ tells us how A reacts upon receiving the input event x in configuration (σ,ε). This reaction yields, due to non-determinism, all possible subsequent configurations (σ',ε') together with the output event y. Here, $(\varepsilon,x) \models t$ is true iff the trigger t can be evaluated to true with respect to the valuation ε and the current event x; $(Y \mapsto 8, \{a,b\}) \models (5 \leq Y) \wedge a$, for example. To join all such pairs in one set, we define:

$$[\![t]\!]_A =_{df} \{(\varepsilon,x) \in \mathcal{E}(A) \times \mathcal{I}(A) \mid (\varepsilon,x) \models t\}$$

The tuple $\mathcal{R}[\![com]\!]\varepsilon$ consists of the next valuation ε' and the output event y that are obtained when the command com is carried out with respect to the current valuation ε. By the predicate $[\![\delta]\!](c,x) = \emptyset$ also chaotic behavior is included in the semantic set $[\![A]\!]$. 'Chaotic' here means that whenever for the automaton A in the current configuration c a transition relation for the current input event x is not defined, it can produce an arbitrary output sequence $y\&o$ and can change to an arbitrary successor configuration c'. Later on, in the design process this underspecification can be reduced; mathematically, this means to transform chaotic behavior in well-defined behavior. To find the greatest solution for the equation $(*)$ is equivalent to find the greatest solution for $F(X_0) = X_0$, i.e. the greatest fixpoint gfp(F) of F, where F is defined by the lambda term

$$F =_{df} \lambda X.\{(c, x\&i, y\&o) \mid \exists c'.((c',y) \in [\![\delta]\!](c,x) \wedge (c',i,o) \in X) \vee [\![\delta]\!](c,x) = \emptyset\}$$

[1] Note that the brackets $[\![.]\!]$ are overloaded.

Notice that $\emptyset$ is always the least fixpoint, if $[\![\delta]\!](c,x) \neq \emptyset$. Least fixpoints yield finite objects, whereas greatest fixpoints are related to infinite solutions. As we deal with infinite I/O histories, we therefore look for the greatest fixpoint gfp(F) which is characterized by

$$\bigcup\{X \in \wp(\mathcal{C}(A) \times \mathcal{I}(A)^{\infty} \times \mathcal{O}(A)^{\infty}) \mid X \subseteq F(X)\}$$

The monotonicity of F is a sufficient condition for the existence of this fixed point.

Proposition 1 *F is a monotonic function with respect to the subset ordering on power sets.*

As F is a monotonic function on a complete lattice (the power domain), a greatest fixpoint always exists (propositions of Knaster/Tarski and Tarski). The semantics of the composition with instantaneous feedback is defined as follows (see also Figure 3 to get a better intuition):

$$\begin{aligned}[\![S_1 \lhd L \rhd S_2]\!] =_{df} \{((c_1,c_2),i,o) \mid \exists o_1,o_2.o_1 \in \mathcal{O}(S_1)^{\infty} \wedge o_2 \in \mathcal{O}(S_2)^{\infty} \wedge \\ o = o_1 \cup o_2 \wedge \\ (c_1, i|_{In(S_1)\setminus(L\cap Out(S_2))} \cup o_2|_{Out(S_2)\cap L\cap In(S_1)}, o_1) \in [\![S_1]\!] \wedge \\ (c_2, i|_{In(S_2)\setminus(L\cap Out(S_1))} \cup o_1|_{Out(S_1)\cap L\cap In(S_2)}, o_2) \in [\![S_2]\!]\}\end{aligned}$$

where $\cup$ here is the pointwise extension of the set-theoretic union on streams of sets and $s|_X$ the pointwise restriction of stream elements (= events) in s to signals in X. Our notion of instantaneous feedback (= feedback in the same instant) was inspired by Argos [8]. It resembles the technique for solving equations in Argos but adds non-determinism and chaotic behavior.

The pure parallel composition $S_1 \| S_2$ of two components S_1 and S_2 is defined as special case: $S_1 \| S_2$ is regarded to be a syntactical abbreviation for $S_1 \lhd \emptyset \rhd S_2$. Just as simple is the definition of signal hiding:

$$[\![[S]_K]\!] =_{df} \{(c,i,o|_{Out(S)\setminus K}) \mid (c,i,o) \in [\![S]\!]\}$$

Having defined the formal semantics for μ-Charts we can discuss some interesting semantical properties of our language:

- Composition is commutative.
- In general, the composition operator does *not* have any associativity-like properties; especially, the following is in general not true:

$$[\![S_1 \lhd L \rhd (S_2 \lhd L \rhd S_3)]\!]_{io} = [\![(S_1 \lhd L \rhd S_2) \lhd L \rhd S_3]\!]_{io}$$

- Other algebraic properties which one would appreciate to be fulfilled but indeed are *false* are, due to non-determinism, redundancy, and distributivity; instead we have:

  - $[\![S||S]\!]_{io} \neq [\![S]\!]_{io}$
  - $[\![(S_1||S_2) \lhd L \rhd S_3]\!]_{io} \neq [\![(S_1 \lhd L \rhd S_3)||(S_2 \lhd L \rhd S_3)]\!]_{io}$
  - $[\![(S_1 \lhd L \rhd S_2)||S_3]\!]_{io} \neq [\![(S_1||S_3) \lhd L \rhd (S_2||S_3)]\!]_{io}$

- In contrast, redundant specifications in general increase the non-deterministic behavior of the system: $[\![S]\!] \subseteq [\![S||S]\!]$. The opposite direction $[\![S||S]\!] \subseteq [\![S]\!]$ generally does not hold for non-deterministic specifications; $[\![S]\!]_{io} = [\![S||S]\!]_{io}$ is true only if S is deterministic.
- Furthermore, if the interfaces of two combined specifications do not fit together, they behave as purely parallel composed: For $In(S_1) \cap L \cap Out(S_2) = In(S_2) \cap L \cap Out(S_1) = \emptyset$ the following holds: $[\![S_1 \lhd L \rhd S_2]\!] = [\![S_1||S_2]\!]$.

4 Specification Refinement

In the previous section we have introduced our automata-oriented specification language. We have defined its semantics, i.e. its input/output-behavior in terms of streams. However, a pure specification formalism is worthless without any system development process. What we need is to know how to develop a concrete implementation or realization from an abstract system specification, that is, how to generate hardware or software from it.

It is usually impossible to carry out this transformation in only one step. In practice, the situation is even worse. For complex systems, even a design specialist may not be capable to write down an abstract specification ad hoc. Rather such a system will be developed by applying subsequent concretization steps, whereby after every single step the overall system behavior is a bit more concrete. Each of these steps is called a *refinement* step. The final implementation then is only the most precise specification that is suitable to run on a certain machine. First ideas on a state-based refinement calculus have been developed in [11, 12].

A specification S_2 is a refinement of another specification S_1 ($S_1 \leadsto S_2$) iff $In(S_1) \subseteq In(S_2)$, $Out(S_1) \subseteq Out(S_2)$ and the following is true:

$$\{(i|_{In(S_1)}, o|_{Out(S_1)}) \mid (i,o) \in [\![S_2]\!]_{io}\} \subseteq [\![S_1]\!]_{io}$$

Notice that there is a good reason to restrict both input and output to $In(S_1)$ and $Out(S_1)$, respectively. Otherwise, one of the most intuitive refinement rules, hierarchical decomposition, would by no means be sound.

We expect our refinement calculus to be stepwise applicable. Therefore, we want to guarantee that also a sequence of refinement steps are a refinement of the original specification again. As $\leadsto$ is transitive, we can guarantee this in our case.

Besides transitivity, compositionality is a further important property for our semantical framework. It secures that whenever a small part of a large specification is refined, also the entire model is refined:

Proposition 2 *If $S_1 \leadsto S_2$ then also $S_1 \lhd L \rhd S_3 \leadsto S_2 \lhd L \rhd S_3$ for arbitrary S_3 and L with $L \subseteq Out(S_1) \cup Out(S_3)$ and $[S_1]_{L'} \leadsto [S_2]_{L'}$ for arbitrary L'.*

Notice that in the above proposition it is essential that $L \subseteq Out(S_1) \cup Out(S_3)$ and not $L \subseteq Out(S_2) \cup Out(S_3)$. Otherwise, as $Out(S_1) \subseteq Out(S_2)$, S_2 could possibly perform additional behavior due to extra communication that is not possible with S_1.

4.1 The Calculus

In this section, we give a set of purely syntactical rules whose application guarantees the software engineering specialist correct refinement steps. She or he does not need to be aware of the formal semantics but just has to apply the intuitive syntactical rules in a correct way. Hence, the stepwise refinement within a calculus for μ-Charts is not only a mathematically appealing idea but also a realistic procedure to be applied in an industrial environment.

Rules for Sequential Automata. In principle, to show that a sequential automaton A_2 is a refinement of another sequential automaton A_1, where $A_k =_{df} (N_k, I_k, O_k, \Sigma_k, \Sigma_{0k}, V_{lk}, \varphi_{0k}, \delta_k)$ we have to show that $\text{gfp}(F_{A_2}) \subseteq \text{gfp}(F_{A_1})$. This proof obligation can, whenever both A_1 and A_2 have the same interface and the same configurations, be relaxed to (†):

$$\forall X \subseteq \mathcal{C}(A_1) \times \mathcal{I}(A_1)^\infty \times \mathcal{O}(A_1)^\infty : X \subseteq F_{A_2}(X) \Rightarrow F_{A_2}(X) \subseteq F_{A_1}(X)$$

With this preliminaries we now can present the refinement calculus for automata:

Remove Initial States. The reduction of the initial states Σ_0 to $\Sigma_0' \subseteq \Sigma_0$ is a correct refinement step. In our example, we can reduce the initial states from {UNLD, LOCKED} to {UNLD}.

Add Additional States. The set of states Σ of a sequential automaton can be enlarged by Σ' and the semantics keeps exactly the same as long as the initial states are not modified and the fresh states are not "connected" to the rest of the automaton with already existing transitions. In subsequent refinement steps, however, theses states can be connected with fresh transitions according to the following transition rules. Notice that this rule merely allows to add new states on the same hierarchical level. Whenever new states on an hierarchical different level shall be added, the rule for hierarchy (see below) has to be applied.

In the sequel, let A_i be the automaton $(N, I, O, \Sigma, \Sigma_0, V_l, \varphi, \delta_i)$, for $i = 1, 2$, i.e. A_1 and A_2 only differ in their transition relations.

Delete Transitions. If we obtain A_2 from A_1 by deleting the transition (t, com, σ') from $\delta_1(\sigma)$, i.e. $\delta_1(\sigma) = \delta_2(\sigma) \cup \{(t, com, \sigma')\}$, this is a correct refinement step if $t \Rightarrow \bigvee_{t' \in T_{\delta_2}(\sigma)} t'$ is a tautology, where $T_\delta(\sigma)$ yields the first projection, the trigger condition, of $\delta(\sigma)$. The premise here means that the deleted condition t is already subsumed in the remaining conditions, and therefore, additional non-determinism cannot occur when deleting the corresponding transition[2].

Add Transitions. If we obtain A_2 from A_1 by adding the transition (t, com, σ') to $\delta_1(\sigma)$, i.e. $\delta_2(\sigma) = \delta_1(\sigma) \cup \{(t, com, \sigma')\}$, this is a correct refinement step if $\forall t' \in T_{\delta_1}(\sigma) : t \wedge t'$ is a contradiction. Informally, this premise guarantees that no transitions are introduced, whose triggers are already subsumed in any other existing transition. We want to exemplarily carry out the proof for this rule:

Proof. According to (†), we take an arbitrary $X \subseteq \mathcal{C}(A_1) \times \mathcal{I}(A_1)^\infty \times \mathcal{O}(A_1)^\infty$ with $X \subseteq F_{A_2}(X)$. Now let the following be true: $\exists c'.((c', y) \in [\![\delta_2]\!](c, x) \wedge (c', i, o) \in X) \vee [\![\delta_2]\!](c, x) = \emptyset$. As we must prove $F_{A_2}(X) \subseteq F_{A_1}(X)$ we have to show that $\exists c'.((c', y) \in [\![\delta_1]\!](c, x) \wedge (c', i, o) \in X) \vee [\![\delta_1]\!](c, x) = \emptyset$. This is done by case distinction:

1. First, we assume that $\exists c'.(c', y) \in [\![\delta_2]\!](c, x) \wedge (c', i, o) \in X$. Let $c = (\sigma, \varepsilon)$. We define $\delta_3(\sigma) =_{df} \{(t, com, \sigma')\}$. Since $[\![\delta_2]\!](c, x) = [\![\delta_1]\!](c, x) \cup [\![\delta_3]\!](c, x)$ the tuple (c', y) must either be in $[\![\delta_1]\!](c, x)$ or in $[\![\delta_3]\!](c, x)$. If $(c', y) \in [\![\delta_1]\!](c, x)$ the proof is already completed. Otherwise, $(c', y) \in [\![\delta_3]\!](c, x)$ and so $(\varepsilon, x) \in [\![t]\!]_{A_2}$ must hold. From this we can deduce that $\forall t' \in T_{\delta_1}(\sigma) : (\varepsilon, x) \notin [\![t']\!]_{A_1}$ because $\forall t' \in T_{\delta_1}(\sigma) : t \wedge t' = \text{ff}$ and therefore
$$\forall t' \in T_{\delta_1}(\sigma) : [\![t]\!]_{A_1} \cap [\![t']\!]_{A_1} = [\![t \wedge t']\!]_{A_1} = [\![\text{ff}]\!]_{A_1} = \emptyset$$
As a consequence, we get $[\![\delta_1]\!](c, x) = \emptyset$ what yields the desired result.
2. Second, we assume that $[\![\delta_2]\!](c, x) = \emptyset$. As $\delta_1(\sigma) \subseteq \delta_2(\sigma)$ implies $[\![\delta_1]\!](c, x) \subseteq [\![\delta_2]\!](c, x)$ also $[\![\delta_1]\!](c, x) = \emptyset$ holds. □

The first case of the proof says that a new transition only can make the specification more precise, but not more chaotic. The second case guarantees that chaotic behavior of A_2 must already have been chaotic in A_1.

Modify Existing Transitions. Let A be as above, σ a state in A, and $e \in \delta(\sigma)$ the transition to be modified; δ and δ' denote the transition relations before and after one transformation, respectively. We then can identify the following rules:

1. The trigger condition t, where $e = (t, com, \sigma')$, σ' not necessarily different from σ, can be refined:

[2] Remember that after the application of a refinement rule, the specification only can be more concrete, but not more abstract.

- To $t \vee t'$ if $\forall t''.t'' \in T_{\delta}(\sigma) \Rightarrow (t' \wedge t'')$ is a contradiction
- To $t \wedge t'$ if $(t \wedge \neg t') \Rightarrow \left(\bigvee_{t'' \in T_{\delta'}(\sigma)} t''\right)$ is a tautology

for an arbitrary Boolean term t' in $Bexp(I + V_l)$.

2. The trigger condition $a \wedge a'$, where $e = (a \wedge a', com, \sigma)$, can be refined to a for an arbitrary Boolean term a' in $Bexp(I + V_l)$ if $\forall t''.t'' \in T_{\delta}(\sigma) \Rightarrow (t'' \wedge (a \wedge \neg a'))$ is a contradiction.
3. The trigger condition $b \vee b'$, where $e = (b \vee b', com, \sigma)$, can be refined to b for an arbitrary Boolean term b' over $Bexp(I + V_l)$ if $b' \wedge \neg b \Rightarrow \left(\bigvee_{t'' \in T_{\delta'}(\sigma)} t''\right)$ is a tautology.

Rule number (1) can be proven from the rules for adding and deleting transitions. Rules (2) and (3) are deduced from (1) when t is substituted by $a \wedge a'$ and $b \vee b'$, respectively and the following equivalences are used:

$$a = a \wedge (\neg a' \vee a') = (a \wedge \neg a') \vee (a \wedge a')$$
$$b = b \vee (b' \wedge \neg b') = (b \vee b') \wedge (b \vee \neg b')$$

In the example we can e.g. modify the transition trigger *ldn* in MOTORLEFT to $ldn \wedge \neg lup$. As $lup \wedge ldn \Rightarrow ldn \vee (lup \wedge \neg ldn)$ is a tautology, this is a correct refinement step. If we wanted to refine the remaining transition with label *ldn* to $ldn \wedge \neg lup$, too, we would violate a syntactical refinement condition because $ldn \wedge lup \Rightarrow (ldn \wedge \neg lup) \vee (lup \wedge \neg ldn)$ is no longer a tautology. Finally, we would like to mention that none of the above rules depends on transition commands, but only on trigger conditions.

Rules for Composition. Single components can be composed to more complex specifications using the following rules:

- $S_1 \rightsquigarrow S_1 \lhd L \rhd S_2$ for $Out(S_1) \cap Out(S_2) = \emptyset$ and $In(S_1) \cap L \cap Out(S_2) = \emptyset$
- $S_1 \rightsquigarrow S_1 \| S_2$ and $S_2 \rightsquigarrow S_1 \| S_2$ for $Out(S_1) \cap Out(S_2) = \emptyset$ as direct consequence from the first rule.

Informally, these rules express that S_1 can be composed with any other specifications S_2 whenever S_2 cannot add additional behavior due to message sending to S_1 and the output interfaces are disjoint. If the latter condition would be violated, S_2 could chatter in the output stream of S_1 and one could not distinguish anymore whether events are generated by S_1 or S_2. Again, additional non-determinism possibly would be introduced.

As a consequence, in our running example $S_L \rightsquigarrow S_L \| S_R$, $S_R \rightsquigarrow S_L \| S_R$, and also $S_C \rightsquigarrow S_C \lhd L \rhd (S_L \| S_R)$ and $S_L \| S_R \rightsquigarrow S_C \lhd L \rhd (S_L \| S_R)$, where S_L, S_R, and S_C denote the left and right motor, and the control part of the central locking system, respectively. Due to the transitivity of $\rightsquigarrow$, we also get $S_L \rightsquigarrow S_C \lhd L \rhd (S_L \| S_R)$. Note that, for lack of associativity, we cannot omit brackets as $rdn, rup, ready \in Out(S_C) \cap L \cap In(S_R)$ and $rmr \in In(S_C) \cap L \cap Out(S_R)$.

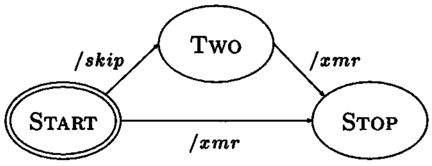

Figure 5. Decomposition of DOWN and UP, $x \in \{l, r\}$

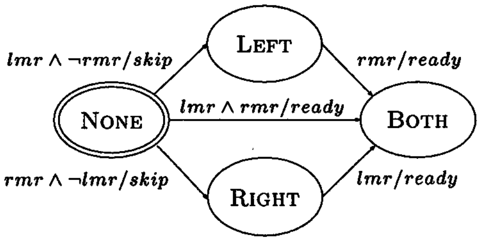

Figure 6. Decomposition of LOCKG and UNLG

Rule for Hierarchy. In the last section we have seen that hierarchical composition can be interpreted as parallel composition plus some extra communication of master and slaves. From the propositions for composition we therefore can deduce the following rule:

Let $A\,\mathsf{decby}\,(\Sigma_d, \varrho)$ be defined as in Section 3.1. Then we get

$$A \leadsto A\,\mathsf{decby}\,(\Sigma_d, \varrho)$$

without any further restrictions in case of strong preemption. For weak preemption this proposition is only true, if

$$\forall k.(1 \leq k \leq |\Sigma_d| \Rightarrow O \cap Out(S_k) = \emptyset = I \cap Out(S_k))$$

where $\{S_1, \ldots, S_{|\Sigma_d|}\} = \bigcup_{\sigma \in \Sigma_d} \varrho(\sigma)$. This rule can be easily derived from the refinement rules for composition, the definition of hierarchy, and the fact that $\{go(\sigma)\} \cap In(A) = \emptyset$ as $go(\sigma)$ is a fresh signal. If we take a look at the central locking system, we ascertain that the automaton in Figure 5 is a correct refinement of DOWN and UP in LEFTMOTOR and RIGHTMOTOR, respectively.

Figure 6 however is no correct refinement of UNLG and LOCKG because $ready \in Out(\text{UNLG}) \cap In(\text{CONTROL})$ and $ready \in Out(\text{LOCKG}) \cap In(\text{CONTROL})$. We see that the restriction $I \cap Out(S_k) = \emptyset$ is really needed as $A\,\mathsf{decby}\,(\Sigma_d, \varrho)$ can be embedded in a specification that makes certain signals available for communication which would lead to additional non-determinism. In the example, the signal *ready* is fed back on the outermost level of hierarchy, which could pander self termination. Therefore, we can conclude that to introduce self termination never is a correct refinement step.

5 Conclusion

Reactive systems are often part of safety critical systems. To obtain correct working systems that do not damage or destroy its environment, it is important to keep the correct design of such systems in eye from the very beginning. One possibility to reduce the number of critical malfunctions is to apply formal verification techniques, such as model checking.

It is hardly possible to specify complicated systems ad hoc. Hence, in a typical design process the designer starts with a first draft, which is later on transformed step by step into a more and more complex system. As a consequence, critical errors can be included in any design stage. Clearly, fully or semi automated verification techniques help to find out many unwanted behaviors before the system is implemented. However, many malfunctions could be avoided if the designer had a design methodology by hand that prevented him to specify unwanted behavior. One part of such a methodology is a set of rules that tells the user which transformations of the original specification are allowed.

In this paper we have proposed a refinement calculus for a synchronous Statecharts dialect that makes a contribution to this task. We have shown that two syntactical constructs are enough to formulate Statecharts specifications. We have described the semantics of μ-Charts mathematically and have described how the notion of refinement can smoothly be integrated in this semantics.

Further work will focus on the question how our refinement calculus can be embedded as part in a more general design methodology for Statecharts.

Acknowledgment

I would like to thank Jan Philipps and Bernhard Rumpe for many fruitful discussions on this and on related topics and for the careful reading of a draft version of this paper. I am also grateful for the inspiring comments of the anonymous referees.

References

1. G. Berry. Preemption in Concurrent Systems. In *Foundations of Software Technology and Theoretical Computer Science : 13th Conference Bombay, India, December 15-17*, volume 761 of *Lecture Notes in Computer Science*, pages 72 – 93. Springer, 1993.
2. G. Berry. *A Quick Guide to Esterel.* Unpublished Esterel Primer, 1996.
3. G. Berry. 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, 1998.
4. M. Broy. Interaction Refinement - The Easy Way. In *Program Design Calculi*, volume 118 of *NATO ASI Series F: Computer and System Sciences.* Springer, 1993.

5. D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231 – 274, 1987.
6. D. Harel and A. Naamad. The Statemate Semantics of Statecharts. *ACM Transactions On Software Engineering and Methodology*, 5(4):293–333, 1996.
7. F. Maraninchi. Operational and Compositional Semantics of Synchronous Automaton Compositions. In W.R. Cleaveland, editor, *Proceedings CONCUR'92*, volume 630 of *Lecture Notes in Computer Science*, pages 550 – 564. Springer-Verlag, 1992.
8. F. Maraninchi and N. Halbwachs. Compositional Semantics of Non-deterministic Synchronous Languages. In Riis Nielson, editor, *Programming languanges and systems - ESOP'96, 6th European Symposium on programming*, volume 1058 of *LNCS*. Springer-Verlag, 1996.
9. J. Philipps and P. Scholz. 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, 1997.
10. J. Philipps and P. Scholz. 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, 1997.
11. B. Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme (in German)*. Herbert Utz Verlag Wissenschaft, München, Ph.D. Thesis, Technische Universität München, 1996.
12. B. Rumpe and C. Klein. Automata Describing Object Behavior. In H. Kilov and W. Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, pages 265–286. Kluwer Academic Publishers, 1996.
13. M. von der Beeck. A Comparison of Statecharts Variants. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Proc. Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'94)*, volume 863 of *Lecture Notes in Computer Science*, pages 128 – 148. Springer, 1994.