Code Generation
and Model Driven Development
for Constrained Embedded Software

Andrzej Wąsowski

FIRST PhD School
Department of Innovation
IT University of Copenhagen

A dissertation submitted to IT University
in partial fulfillment of the requirements for the degree
of doctor of philosophy in Computer Science

Copenhagen, January 31, 2005

Code Generation and Model Driven Development
for Constrained Embedded Software

Department of Innovation
IT University of Copenhagen
Rued Langgaard Vej 7
2300 Copenhagen S
Denmark

Abstract

We consider statechart models of discrete control embedded programs operating under severe memory constraints. There have been very few results in code generation for such systems. We analyze code generation methods for embededded processors utilizing C as an intermediate language and runtime interpreters. We choose a suitable subset of hierarchical statecharts and engineer an efficient interpreter for programs in it. An algorithm is provided that simplifies general models to our sublanguage removing *dynamic scoping* and *transition conflicts*. The resulting code generator improves over an industrial implementation provided by IAR A/S.

The interpreter for hierarchical statecharts is complex. We define *flattening* as a process of transforming hierarchical models into their hierarchy-less counterparts. We prove that even with a simulation-based correctness criterion any flattening algorithm would cause a *super polynomial growth of models*, if it does not exploit message passing. Then we devise a *polynomial flattening algorithm* based on internal asynchronous communication in the model. The implementation of this algorithm beats our earlier hierarchical code generator by 20–30% on realistic examples.

In the second part of the thesis we develop a unified theory for specifying correctness of model transformations and modeling software product lines. Our framework is based on a novel notion of *color-blindness*: a dynamically changing inability of the environment to observe differences in system outputs. Being safe approximations of all possible usage scenarios such environments can be used to specify specialized versions of the product. We propose a correctness criterion for specialization algorithms based on Larsen's relativized bisimulation extended with color-blindness.

Any good modeling formalism for software product lines supports composition and step-wise modeling, so that families can be organized in hierarchies or even more flexible structures. To serve this purpose we introduce an *information ordering* on our models of environments. Crucially, we show that the abstract information preorder can be *characterized operationally* by means of simulation. Then we use the information preorder to define intuitive *composition operators* as meets and joins in the associated quotient lattice. We demonstrate an extended example using a hierarchical family of alarm clocks specified by means of color-blind environments.

Preface

The work presented in this thesis has been performed during PhD studies at the IT University of Copenhagen from August 2001 until February 2005, while I participated in the *Resource Constrained Embedded Systems* project in the Department of Innovation (Danish National Research Councils grant no. 2051-01-0010). I would like to thank the head of the project and my principal supervisor, Peter Sestoft, first for accepting me as a student, then for all the time devoted to my work, and for the gentle and understanding supervision that allowed a lot of my independence. Last but not the least, for reading and commenting on all my papers.

The main person working in that period was my wife. Ola not only did everything to let me focus on my project, but also gave birth to and looked after our two sons Karol and Jakub. I am tremendously grateful to all her devotion and support, without which I could not possibly succeed. I have to apologize to both boys for not devoting enough time to them, especially to Karol, whose growth and development proceeded literally in parallel to the development of this work. Fortunately, now after three years, I experience that his advances are much more impressive than mine. I also apologize to my parents and sisters, and all friends and relatives in Poland, that had to put up with our long absences from Warsaw in the past three years.

The *Center for Embedded Software Systems* at Aalborg University became my second home during my studies. It was possible due to enormous hospitality of its leader Kim G. Larsen, who invited me to stay with CISS and supervised me during my stay. The theory presented in chapters 5–6 emerged during common meetings with him and Ulrik Larsen. Ulrik has also contributed significantly as a helper editor of our paper on this topic, which unavoidably leaves traits of his work in this thesis. Some aspects of the flattening algorithm presented in chapter 4 are inspired by an unpublished work of Gerd Behrmann of CISS. The implementation of charter, the Java code generator mentioned in chapter 3 was made jointly with Jørgen Steensgaard-Madsen of DTU, the author of Dulce. I would also like to thank Henrik Hulgaard (of ITU and ConfigIt), for helping me in early months of my studies, whenever Peter was not available.

I would like to thank my colleagues, friends, teachers and superiors that have devoted their patience and time to help me (or just made my PhD life

more enjoyable): Alexandru Barlea, Carsten Butz, Martin Elsman, Juhan Ernits, Jan Jürjens, Ken Friis Larsen, Krzysztof Kaczmarski, Kaare Jelling Kristoffersen, Henrik Leerberg, Henning Makholm, Erik van der Meer, Marius Mikucionis, Jørn Lind-Nielsen, George Milne, Brian Nielsen, Henning Niss, Gergely Pinter, Henrik Reif Andersen, Jakob Rehof, Emil Sekerinski, Volodya Shavrukov, Kent Rene Simonsen, Arne Skou, Jiri Srba, Susana Tosca, Christian Worm Mortensen, Albert Zündorf, and Kasper Østerbye. I should not forget any of my office mates from various places and periods: Jens Alsted, Jakob Illum, Alexandre Krivoulets, Francois Lauze, Rasmus Møgelberg, Rasmus Petersen, Henrik Schiøler, Sathiamoorthy Subbarayan, Noah Torp Smith, Jun Yoneyama; and those of ITU students that were brave enough to explore intricacies of statecharts and UML: Lars Bengtsson, Mette Berger, Lone Gram, Karsten Pihl.

Last but not the least, this work would had never started if Bartek Klin had not forwarded me the announcement about the vacant PhD positions at ITU and if my superior in Warsaw, Prof. Dr hab. Bohdan Macukow in the Department of Mathematics and Information Science at Warsaw University of Technology had not let me go on leave to work at IT University.

All my work would become very difficult, if not impossible, without a great mass of open source projects and other freely available tools. I had the pleasure of using at least the following: Acrobat Reader, Buddy, bzip2, coreutils, Cygwin, cpp2latex, CVS, Debian GNU/Linux, ddd, Dia, dulce, epssplit, foilTeX, gcc, gdb, Gentoo Linux, Ghostscript, Gnome, GNU arch, GNU awk, GNU Emacs, GNU grep, GNU make, GNU sed, GNU screen, GNU tar, Gnuplot, Graphviz, GV, gzip, fxp, LaTeX, Linux, Midnight Commander, METAPOST, MLton, Moscow ML, Mosmake, Mozilla, Mozilla Firefox, Mozilla Thunderbird, Muddy, Perl, psutils, Ratpoison, RedHat Linux, semantics, SML/NJ, Sun Java compiler, TeX, Vim, Xfig, xfree86, xindy, xorg-x11, and xpdf.

The Danish division of IAR Systems has provided me with tools for embedded development: IAR visualSTATE, Embedded Workbench and a collection of embedded compilers. I-Logix provided Rhapsody in Micro C under conditions of their academic program. I had the opportunity to study parts of implementations of thermostat controllers, kindly provided by Danfoss within the EKC project at Aalborg University.

I apologize everybody that I have omitted. I thank you all and hope that what you find in this thesis at least partly rewards your help and support.

Contents

List of Figures

List of Tables

List of Symbols

$\mathcal{P}(X)$	the set of all subsets of X (a power-set)
$\mathcal{M}(X)$	the set of all multisets (bags) containing elements of X
X^*	the set of all finite sequences of elements of X
$X \times Y$	cartesian product (the full relation) of sets X and Y, also a product of environments, see section 5.3, p. 114
$X \hookrightarrow Y$	the set of all partial functions from X to Y
$X \to Y$	the set of all total functions from X to Y
$f\big|_X$	function f restricted to domain X, $X \subseteq dom(f)$
$dom(f)$	the domain of function f
$rng(f)$	the range of function f
$\varrho[v/x]$	substitution: v is the new value of function ϱ for x
$\pi_n(x)$	the nth component of a list or a tuple (nth projection)
$a\,R\,b$	a is in relatation R with b: $(a, b) \in R$
R^*	if R is a relation: the reflexive transitive closure of R
R^+	the irreflexive transitive closure of relation R
$X \uplus Y$	disjoint union of multisets (bags) X and Y
$a \sqcup b$	the least upper bound (lub) of a and b
$a \sqcap b$	the greatest lower bound (glb) of a and b
$\bigsqcup X$	the least upper bound of elements of set X
$\bigsqcap X$	the greatest lower bound of elements of set X
$\langle a_1, \ldots, a_n \rangle$	a list (sequence) consisting of elements from a_1 to a_n
$\langle \rangle$	an empty list
$elems(L)$	set of all elements of list L
$|L|$	length of list L
$L \upharpoonright X$	list created from L by removing elements not in set X
$[a; b]$	a closed interval of real numbers between a and b
[a..b]	a visualSTATE type denoting integers in $[a; b]$, see p. 13
out-degree	number of edges out-going from a vertex in a graph

$Event$	set of external input events for a statechart, see p. 14
$Action$	set of outputs of a given statechart, see p. 14
$Signal$	set of internal events in a given statechart, see p. 14
$State_{\mathsf{and}}$	set of all and-states in a given statechart, see p. 11
$State_{\mathsf{or}}$	set of all or-states in a given statechart, see p. 11
Exp	set of all arithmetic expressions, see p. 14
$Aexp$	set of all action expression, see p. 14
$s_1 \perp s_2$	states s_1 and s_2 are orthogonal, see Def. 2.3 on p. 12
$s_1 \not\perp s_2$	same as $\neg(s_1 \perp s_2)$
$NCA(X)$	the nearest common ancestor of states in X, see p. 12
Γ_E	a type environment of events in a statechart, p. 14
Γ_F	a type environment of actions in a statechart, p. 14
$\searrow$	statechart's substate relation, see p. 11
$\searrow^2$	$s_1 \searrow^2 s_2$ iff $\exists s.s_1 \searrow s$ and $s \searrow s_2$, see p. 50
●⟡	initial state marker, see p. 12, 25
◉⟡	initial state marker in history state, see pp. 12, 25
his	a set of history states, see pp. 18, 25
$\tau[\![e]\!]$	the type of expression e, a type oracle see p. 13
$ancest^*(s)$	the set of ancestors of state s (inclusive), see p. 12
$children(s)$	the set of children of state s, see p. 12
en	mapping from and-states to entry actions, see p. 15
ex	mapping from and-states to exit actions, see p. 15
$ini(s)$	default (initial) substate of or-state s, see p. 13
$parent(s)$	parent state of state s, see p. 12
$root$	the top state of statechart's hierarchy, see p. 11
$unop$	ranges over unary operators in C
$binop$	ranges over binary operators in C
$iscope(\sigma, s)$	scope of switching to state s in configuration σ, see p. 28
$scope(\sigma, t)$	scopes of transition t fired in configuration σ, see p. 29
In	in environments the set of inputs, see p. 100
Out	in environments the set of all outputs, see p. 100
$\perp$	empty output (no output), see p. 120
Gen	the set of generator states, see pp. 100, 105
Obs	the set of observer states, see pp. 100, 105
$\mathcal{S}, \mathcal{P}$	usually reserved for names of systems

$\mathcal{E}, \mathcal{F}$	usually reserved for environment names
$\mathcal{B}$	the blind environment, p. 106
$\mathcal{V}$	the perfect vision environment, p. 107
$\mathcal{L}$	a looping system, p. 113
ignore A	observation classes distinguishing everything but the elements of A, see 6.1 p. 136
ignore {}	an environment observing all actions, see p. 136
Interleave $i_1\,i_2$	an environment generating i_1 and i_2 in an alternating fashion, see p. 117
Equiv $o_1\,o_2$	environment that cannot see the difference between outputs o_1, o_2, see p. 117
equiv A	observation classes unable to distinguish any elements of A, but distinguish everyting else, p. 137
s^0	the initial state of an IOATS $\mathcal{S}$
$S, P, E, F, \dots$	in environments typically reserved for generator states
$s, p, e, f, \dots$	in environments typically reserved for observer states
P_e	paritioning of *Out* induced by the observer e, p. 115
$\xrightarrow{!}$	generation transition relation, see pp. 100, 105
$\xrightarrow{?}$	observation transition relation see pp. 100, 105
$X \leqslant Y$	process X simulates Y, see p. 5.2
$X \leqslant_Y Z$	X simulates Z in environment Y, see p. 102
$X \sim Y$	X is equivalent to Y (bisimulation), see p. 101
$X \lesseqgtr Y$	X is equivalent to Y (two way simulation), p. 121
$X \sim_Y Z$	X and Z are equivalent in Z (bisimulation), see p. 103
$X \lesseqgtr_Y Z$	X and Z equivalent in Z (two-way simulation), p. 129
$\mathcal{E} \sqsubseteq \mathcal{F}$	$\mathcal{F}$ is more discriminating than $\mathcal{E}$, by means of relativized simulation, see p. 110
$\mathcal{E} \overline{\sqsubseteq} \mathcal{F}$	$\mathcal{F}$ is more discriminating than $\mathcal{E}$, by means of two way relativized simulation, see p. 129
$\mathcal{E} \underset{\sim}{\sqsubseteq} \mathcal{F}$	$\mathcal{F}$ is more discriminating than $\mathcal{E}$, by means of relativized bisimulation, see p. 129
$\sum X$	sum of observers (generators) in X, see p. 115
$\prod X$	product of observer (generators) in X, see p. 115, in statecharts semantics: an output constructor, p. 21
$e \otimes f$	product of DFA classifiers e, f, see p. 126

List of Terms

action a discrete output of the system towards the environment, an atomic part of system's response to enviornment's event. See p. 14.

and-state a composite statechart state, comprising zero or more concurrent components. See p. 11.

blind environment the environment that is not able to distinguish any two systems. See p. 106.

bisimulation a classic equivalence relation. See p. 103.

color-blindness a dynamic property of a transition system that it cannot distinguish between certain kinds of outputs. See p. 105.

configuration a set of active states of a statechart. See p. 18.

DFA classifier a finite automaton used for classifying regular words into separate sets (corresponding to states of the automaton).

discrimination preorder a preorder on IOATS induced by relativized simulation: one environment is more discriminating than another iff it can distinguish more systems by means of relativized simulation. See p. 110.

generator in IOATS a state capable of producing an output. See pp. 100, 105.

entry action an action executed each time its owner state becomes active. See p. 15.

exit action an exit action executed whenever its owner states becomes inactive. See p. 15.

event a discrete stimulus provided by an environment (an external event) to the system. In visualSTATE events may carry values. See p. 14.

generation relation a transition relation of an IOATS that describes its ability to produce outputs. See pp. 100, 105.

history state an or-state that preserves its active substate in the same fashion as static variables in C preserve their values across function calls. Upon activation or-state activates the one of its children, which was active upon the most recent deactivation. See p. 12, 25

initial and-state the starting state of a given component (or-state). This is the state that is activated when the component is activated. See p. 12, 25

input-enabledness ability to accept any input at any given point in time. Input enabled transition systems are *non-blocking*. See pp. 32, 100

iscope a scope of a state change from some configuration to a single state. See p. 28.

NCA the nearest common ancestor of two or more states. See p. 12.

observer in IOATS a state capable of receiving inputs. See pp. 100, 105.

observation relation a transition relation of an IOATS describing its ability to accept inputs (and react to them). See pp. 100.

or-state a composite statechart state comprising one or more sequentially related and-states (a state machine). See p. 11.

orthogonal states two statechart states are orthogonal if they can be active at the same time (belong to concurrent parts of the model). See p. 12.

out-degree the number of edges out-going from a vertex in a directed graph.

perfect vision environment a universal environment that is able to distinguish any two systems of given sort. See p. 107.

relativized bisimulation bisimulation restricted to executions which can be provided by a given environment (both compared systems are embedded in the same environment before establishing the equivalence). See p. 103.

relativized simulation Simulation restricted to executions which can be provided by a given environment (both the refining and the refined system are embedded in the same environment). See p. 102.

relativized two-way simulation Two-way simulation restricted to executions which can be provided by a given environment (both systems are embedded in the same context, before the equivalence is considered). See p. 129.

scope a set of or-states bounding the parts of the model that will be modified by a given transition in a given state configuration. A set of *iscopes* for the targets of this transition. See p. 29.

signal an internal event provided by the system to itself, or more precisely by one of the components of the system to the other. Signals have higher priority than events. Signals are dispatched gloabally, but may be ignored by some components, and processed by others in the same style as external events are. See p. 14.

simulation a classic notion of refinement, requiring the the refined system must be able to mimick the behaviour of the refining system. See p. 100.

state marker a flag indicating some property of a state (for example initial states, history states).

two-way simulation an equivalence relation induced be the simulation refinement preorder. See p. 121.

1

Introduction

The world around us is full of electronic devices. Modern microwave ovens, coffee makers, vending machines, radio receivers, telephones, washing machines, TV sets, light switches, hearing aids, door locks, lifts, refrigerators, and fire detectors are controlled by microcontrollers. The number of these special purpose computers around us has long ago exceeded the number of personal computers.

The total sales of personal computers are reported to be below 200 million items annually (after news.com). At the same time just one of the major suppliers on the microcontroller market, Microchip Technology, reports that they have sold a billion PIC microcontrollers within two years (2002–2003). The total number of PIC microcontrollers sold exceeds three billion chips now. Similarly another important vendor, Motorola celebrated the sales of 5 billionth 68HC05 microcontroller already in 2001. Needless to say, embedded systems gain more and more attention from the academic community. This dissertation is concerned with the programs running on these small devices, commonly referred to as *embedded systems*, the programs themselves dubbed *embedded software*.

What are the specifics of the embedded software that distinguish it from any other kind of software? Shortly speaking: *high reliability*, *prohibitive cost of upgrades*, and production in *multiple variants*. We have to trust that the embedded software is correct: otherwise it may put our lives at risk. Managers do expect that embedded software is correct: the cost of upgrade is unacceptably high—in many cases it amounts to withdrawal of all the devices sold from the market. Embedded systems are produced in multiple versions to meet the various market needs. Most variants differ by selection of features and their price.

Embedded programs are control programs: they indefinitely accept stimuli from their environment via *sensors* and control the environment by responses given to *actuators*. This is what makes them different from *non-interactive* data processing programs such as compilers. Embedded pro-

grams are thus best described [38] as *reactive* and *synchronous* [46]: always ready to accept a stimulus and infinitely quickly producing a response to the environment. Numerous embedded programs—for example all media applications—are highly concerned with data processing, too. In this thesis we ignore the data processing issues. We focus on discrete control specifically. Discrete control systems are typically the simplest and the smallest of all the devices, and as such they exhibit the hardest resource constraints.

We are interested in highly constrained embedded systems, which only have few resources available. In particular, they are restricted in the size of available memory. It is a fact that quite a few of the embedded devices have only a minuscule amount of writable memory. Some readers may be surprised to discover this, especially if they are used to vast resources available in personal computers.

The limitations in the size of memory are caused by several factors. More memory uses more power, so battery-operated devices tend to be restricted. More memory uses more *physical* space, while the smallest devices strive to minimize the size. The requirements for applications grow much faster than memory prices fall. There is a pressure to deliver on currently available hardware. Producers want to make the maximum use of the resources paid for. Once the amount of memory in the device is set, they still strive to fit as much functionality in it as possible. Less memory costs less money. With embedded devices, often being produced in hundred thousands a year, a saving of a single US dollar per item is no longer negligible from the management's perspective. Especially if it can be achieved *without* sacrificing the user experience. Software divisions make efforts to put as low requirements on hardware as possible.[1]

Last but not the least, memory is getting cheaper. Some would say it is cheap already. Everything depends on the point of reference, though. Memory is, perhaps, cheap relative to the total value of a workstation, but not when compared to the cost of a small light switch. In summary: we want our devices to be richer in features, smaller, more portable and cheaper to produce. Because of that, it is very unlikely that we will ever stop saving on memory in the embedded market.

Let us mention two actual cases that exemplify the above-mentioned trends: wireless sensor networks and small hardware architectures. Bushfire prevention projects are considering distributing wireless fire sensors over large forested areas. A sensor is to be placed every several meters, by simply dropping them from airplanes. Each device should be equipped not only with a fire sensor, but also with a complete wireless communication setup. Needless to say the authorities responsible for the project would prefer that the price of a single item could be measured in cents rather than dollars. Technology needed for making this a reality is emerging [107].

---

[1]Confirmed by informal communications with software divisions of several vendors.
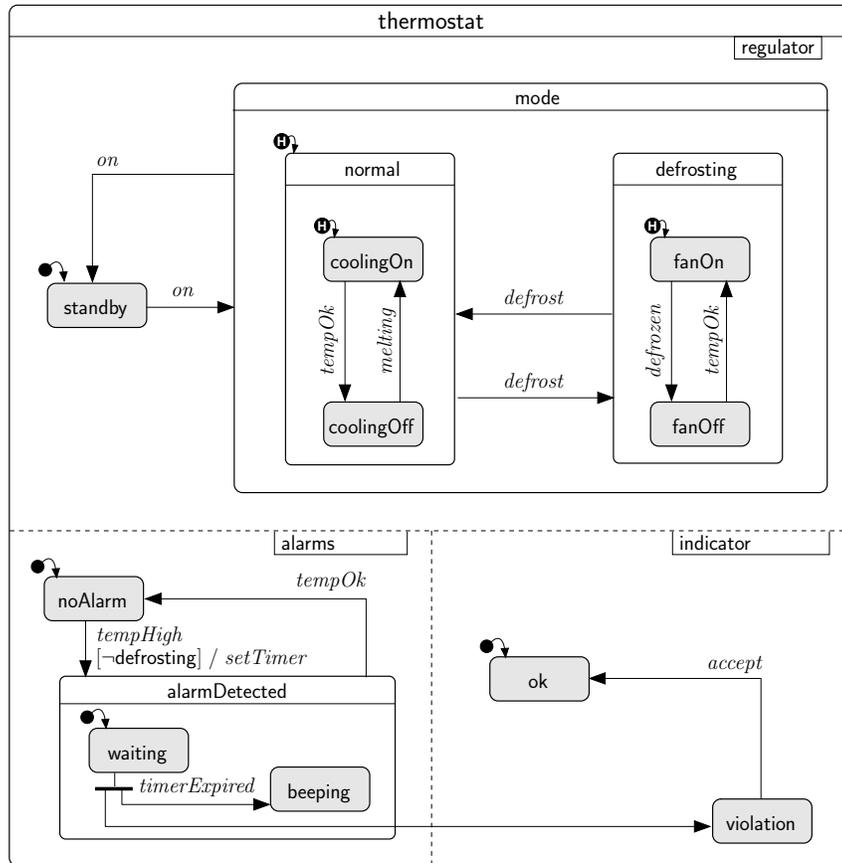
Figure 1.1: A simple abstract model of a thermostat controller.
Disclaimer: This model is a reminiscence of the Danfoss EKC model presented in
[71], which in turn was an academic mock-up model. As such it does *not* reflect the
logics and the complexity of the actual thermostat controllers supplied by Danfoss.

While major CPU vendors on the PC market are constantly racing for
higher frequencies and bigger word lengths, the embedded market is seeing a
wave of slower and smaller architectures. The recent example is the MARC4
line of microcontrollers by Atmel [22]. MARC4 is a 4-bit RISC unit, with
power consumption kept below 1 mA, explicitly targeting wireless applica-
tions. Most of the microcontrollers in this line are equipped with 8 kilobytes
of EEPROM memory (read-only from a typical program's point of view) and
256 4-bit memory cells of RAM. This means *128 bytes* of writable memory!

1.1 The Language of Statecharts

In 1983 [43] David Harel proposed a visual language, statecharts [42], for
modeling reactive synchronous systems. Later in the thesis, we shall define

and explain this language in detail. For now, let us sketch its main properties in an informal manner.

Figure 1.1 presents a simple model of a thermostat controller for a cooling device. We can see that the top state thermostat (the outermost rectangle) is divided in three independent regions separated by dashed lines. These regions, regulator, alarms, and indicator, operate concurrently and synchronously.
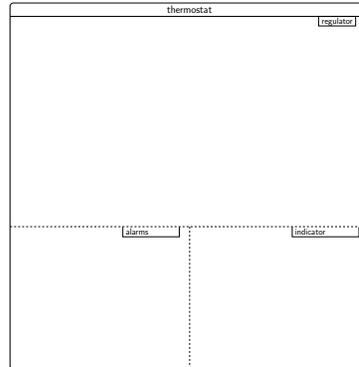
The regulator region contains a state machine responsible for regulator modes:
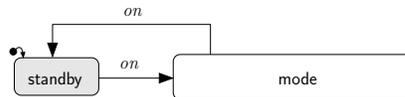
Initially this component (and the entire device) is in the standby mode, which is indicated by a ● marker attached to the respective state. The device can be turned on by pressing the *on* button, modeled as an event over a transition arrow, which makes it enter one of its two major modes.

The first time the device is turned on, it enters the normal mode, which is indicated by a ● marker attached to the respective state. The mode can be switched back and forth between the normal and defrosting mode, by pressing the *defrost* button. The ● symbols mean that if we switch the device off, it will resume operation in the state in which it was interrupted, once we turn it on again, or more abstractly: whenever the mode state is left (by an *on* transition) and reentered again it will activate the same substates that were active when it was left. If ● marker is used instead, then the component is always reverted to it default state upon activation. States *containing* the ● marker are called *history states*; not the states that are actually marked.
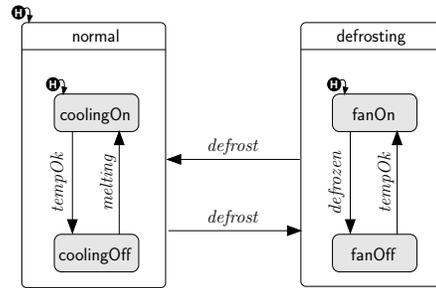
In the **normal** mode the cooler is turned on and off depending on the temperature inside the refrigerator. In the **defrosting** mode the fan, pumping warm air inside the fridge, is turned on, until the temperature reaches the desired level. The mode can be changed by the user, by pressing the *defrost* button.

The other two components, **alarms** and **indicator**, are responsible for handling temperature alarms. If the device is not in the **defrosting** mode (so it is in the **normal** mode) and the temperature inside raises unacceptably high, a violation is detected. A timer is set in order to verify this violation again after a short time. The expression in brackets placed on a transition leaving the **noAlarm** state ([¬**defrosting**]) is called a guard. Guards further restrict possibilities of firing the transition on top of the usual requirements that the event occurs and the source state is active. The *setTimer* label following a slash symbol is an action that executes some code influencing the hardware. In this case it is meant to set up a timer.
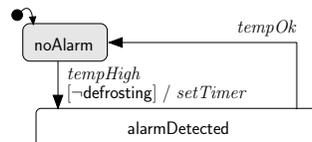
Once the timer expires, and the temperature did not return to its expected level, the alarm is reported by means of turning on a beeper and setting on a **violation** indicator. The beeper turns off automatically as soon as the temperature falls down to the desired level. The **violation** indicator stays turned on for the record, until the operator presses the *accept* button. A transition leaving the **waiting** state (see Fig. 1.1) forks, meaning that it will change both the active state in the **alarms** component and in the **indicator** component.

We have said before that all the components operate concurrently and synchronously: the entire device is able to handle only one event at a time, and all the components react to this event synchronously and concurrently. If the *on* event arrives, the **regulator** component will be the only one undertaking some actions. However if the **coolingOn** is active and so is **alarmDetected** and the *tempOk* event arrives, both **normal** and **alarms** state

will fire their respective transitions simultaneously, activating the coolingOff and noAlarm states respectively. Since statecharts are most often compiled for sequential platforms, this concurrency needs to be sequentialized in some (usually) arbitrary order.

Even without having studied the language of statecharts before, one can immediately appreciate the use of a model in presentation of this control algorithm. A short examination of the model gives a relatively good overview of its works. It would be way more difficult if we had written the same behavior as a C program. Nevertheless, despite numerous success stories, for example the use of models in development in automotive industry and aircraft software design [8], assembly languages and C are still the main implementation dialects in the embedded world (see some examples in [71, 36, 115]). One of the reasons is the memory usage overhead introduced by more abstract approaches.

1.2   Model Driven Development

Inclusion into UML [98] has brought statecharts into the very center of the software engineering realm. Experts claim that models can boost the development process further than just aiding documentation and design. The Model Driven Architecture initiative [99] encourages the use of models at all development stages: from design through implementation, validation and deployment. In case of control models this becomes possible by application of several technologies: model checking [20], test generation, code generation, automatic testing [125], monitoring [49] and specialization [63].

Model checking analyzes models in order to check whether they fulfil initial safety and quality requirements. Automatic test case generation extracts tests from models, fulfilling various coverage criteria. Code generation, the topic of this thesis, translates models to control programs in low-level languages. Automatic testing tests complete program implementations: it checks whether they realize legal behaviors of the model (either by execution of precomputed test cases or referring directly to the model). Monitored execution can detect dangerous situations by observing the software in operation (useful in industrial installations, food storage, ERP applications, etc). Finally specialization, which is closely related to optimizing code generation, helps to obtain several variants of the embedded program from the same source.

Introducing models and abstractions increases the memory consumption of the software. In this thesis we discuss the applicability of model driven development to very constrained embedded programs. We are interested in verifying, whether it is possible to obtain efficient code generation algorithms targeting small devices, for example with 8 or 16 kilobytes of available memory. Following many other researchers, we set off to undermine the reign of

low-level languages on embedded platforms.

We are interested in the most suitable structure of the generated code as well as the applicable optimizations. In the first part of the thesis we analyze both of these questions from theoretical and engineering perspectives. We provide data structures and algorithms for runtime executions and compilation. We implement and evaluate them.

In the second part we focus more on theoretical issues that hopefully will lead to new powerful technologies in future. How can one model a thermostat that does not have a beeper, or does not have a violation indicator? How can we automatically eliminate the functionality of alarms or the entire defrost mode from the model? How do we model the hardware environment of such a system? What assertions can we make about the behavior of the restricted version of our device? In particular, can we guarantee that it does not reach any invalid states? These questions follow from the fact that statechart models are typically *open*: they do not give any information about the behavior of the context in which the program will be embedded. The last two chapters of this thesis are devoted to discussing the form and the power of relevant context models.

1.3   Overview

Let yet another statechart lead us through the developments of this thesis. Figure 1.2 presents a model of a reader. The main state machine of this model (reading) depicts a view of the contents of this thesis. We are now in the introduction chapter. In the *next* chapter, semantics, we shall formally define the language of statecharts, then in chapter 3, code generation, we will state the requirements for code generation, describe the major classes of available methods and relate them to SCOPE, our code generator. Finally we shall focus on the middle layer of SCOPE, model transformations, that simplify the language of the model in order to decrease the size of the generated code. The two transformations discussed are the *elimination of dynamic scoping* and *static conflict resolution.*

Chapter 4 focuses on the back-end of our tool. In this chapter we show that it is possible to implement an efficient code generator, which explicitly preserves the hierarchical nesting of states in the generated code. We demonstrate the algorithms and data structures used and evaluate the results. Our code generator performs better than the reference implementation from the industrial partner. We do not stop there though. We continue to investigate the succinctness gains introduced to the modeling language by means of hierarchical nesting. We find that the saving in size of the model is super polynomial if the language does not contain an internal message passing mechanism. Otherwise the gain is only polynomial and we show an efficient flattening algorithm, which eliminates the hierarchical nesting,
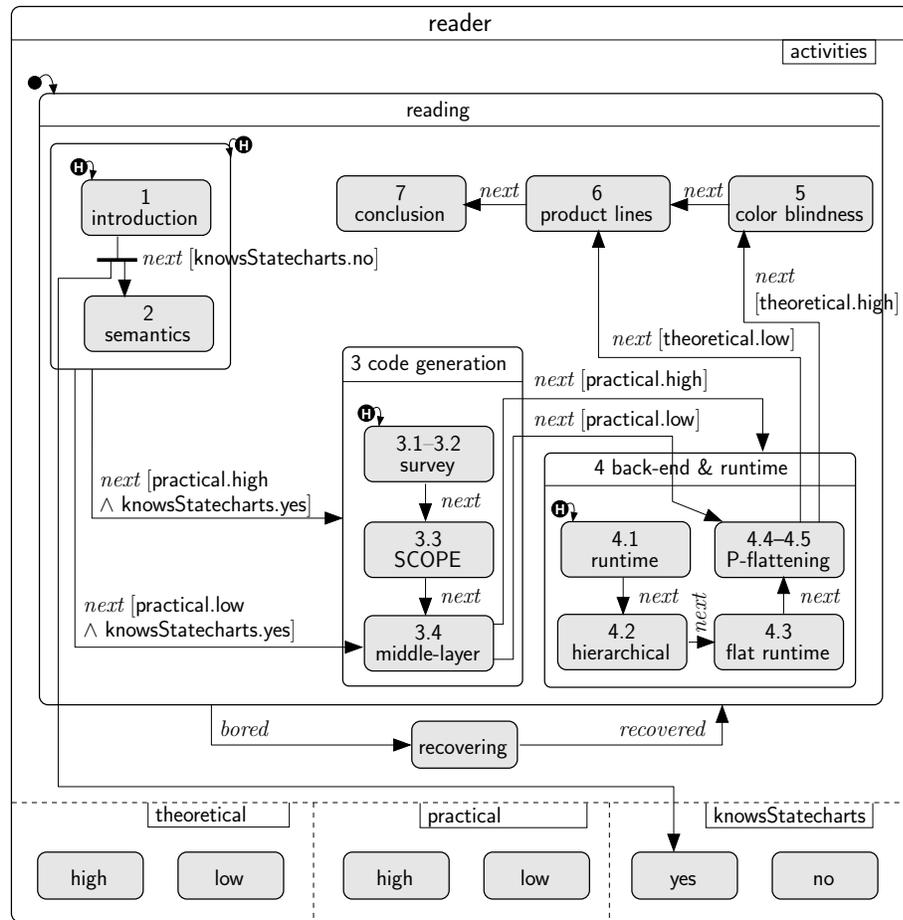
Figure 1.2: A statechart model of the reader of this thesis

and only introduces a polynomial size overhead. This algorithm allows us to implement a new back-end for SCOPE, which beats the hierarchical one and the industrial implementation by up to 80% on some contrived examples. Realistic examples give a gain of 25%.

In the remaining chapters of the thesis we explore the possibilities of supporting product line modeling for reactive synchronous languages. We devise a process algebraic theory of color-blind I/O-alternating transition systems, suitable for modeling limited versions of hardware environments in which our program operates. The language we propose allows hierarchical (stepwise) modeling of product family members. In chapter 6 we demonstrate a product family modeled with our language and discuss the possible obstacles in tool implementation.

The statechart of Figure 1.2 can help you find your own way through numerous sections of this work. Begin with indicating where your interest lies. In case you are highly interested in theoretical aspects of reactive synchronous development, mark the theoretical.high state as active. Otherwise mark it low. In case your interest is more towards engineering aspects, consider the practical region to be set to high. The model assumes that you set at least one of these two areas to high (otherwise, why are you reading so far?). The last configuration region, knowsStatecharts, lets you specify whether you are familiar with the statechart language or not. Once you configured the model you can start reading. Remember to send the *next* event after relevant reading units and evaluate the guards according to your configuration. If you get *bored* you can take a break, entering the recovering state. Once you are *recovered* you can continue reading where you have finished before due to the handy semantics of history states. I wholeheartedly wish you: no deadlocks!

2

The Formal Semantics
of Statecharts

This chapter is devoted to the formal description of the modeling language implemented in the IAR visualSTATE tool [57]. Our goal is not to advance the state of the art in the formal semantics of statecharts, but, rather pragmatically, to give a firm unambiguous definition of the very dialect of statecharts that we experimented with. We discuss syntactic rules, structure of runtime objects and dynamic behaviors by means of big-step operational semantics. Finally we shall cover a multitude of other statechart versions, indicating differences for the most important ones.

An experienced user of statecharts may safely skip this material, returning back when needed. A tool developer may find it interesting for the discussion of various extensions not treated directly in the code generator.

2.1 Static Semantic Model

Let us begin with enumerating the components of every statechart. Somewhat unconventially and imprecisely, we state an incomplete definition first to give an overview, and use the remaining part of the chapter to detail various relations holding among components.

Definition 2.1. *A (hierarchical) statechart is a tuple:*

$$\mathcal{S} = (\mathit{Event},\, \mathit{Signal},\, \mathit{Action},\, \mathit{Var_E},\, \mathit{Var_I},\, \mathit{State},\, \mathit{State_{and}},\, \mathit{State_{or}},$$
$$\mathit{SimpleType},\, \mathit{Type},\, \Gamma_E,\, \Gamma_F,\, \Gamma_V,\, \searrow,\, \mathit{ini},\, \mathit{his},\, \mathit{ex},\, \mathit{en},\, \mathit{Trans})\ \ ,$$

*where Event is a finite set of statechart inputs called events, $\mathit{Signal} \subseteq \mathit{Event}$ is a set of internal statechart events, Action is a set of statechart outputs called actions, $\mathit{Var_E}$ and $\mathit{Var_I}$ are finite sets of external (externally accessible) and internal (not exported) variables respectively, State is a finite set of states, $\mathit{State_{and}}$ and $\mathit{State_{or}}$ are sets of **and**-states and **or**-states forming*
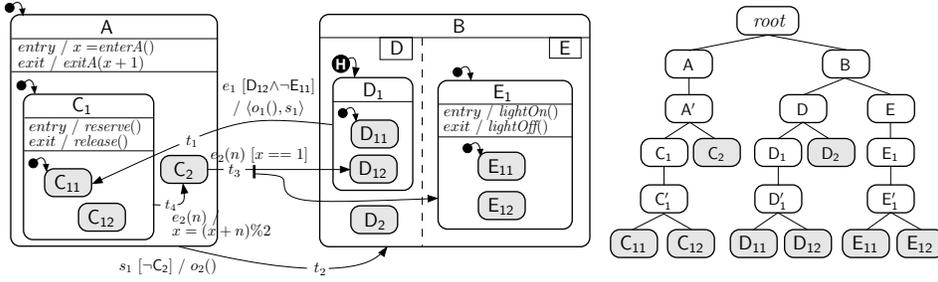
Figure 2.1: A hierarchical statechart and its hierarchy tree

a partition of State, SimpleType is a set of simple types, Type is a set of types, $\Gamma_E : Event \rightarrow SimpleType^*$ is a map defining types of environment events, $\Gamma_F : Action \rightarrow SimpleType \times SimpleType^*$ is a typing of functions, $\Gamma_V : Var_E \cup Var_I \rightarrow Type$ is a typing of variables, $\searrow \subset State \times State$ a sub-state relation, ini : State $\rightarrow$ State is an initial marking, his $\subseteq$ State is the set of history states, ex and en are exit and entry action mappings , and Trans is a finite set of syntactic transitions.

The internal structure of entry/exit action mappings and transitions will be explained later, as soon as all the auxiliary notions utilized by them are introduced.

2.1.1 Hierarchy of States

The sets $State_{\mathsf{and}}$ and $State_{\mathsf{or}}$ form a partition of the general set of states $State$ into two disjoint classes of and-states and or-states respectively:

$$State = State_{\mathsf{and}} \cup State_{\mathsf{or}}, \qquad State_{\mathsf{and}} \cap State_{\mathsf{or}} = \emptyset \qquad (2.1)$$

A distinguished *root* or-state and the substate relation $\searrow \subseteq State \times State$ form a hierarchy of states such that:

$$[\mathsf{or\_root}] \qquad root \in State_{\mathsf{or}} \qquad (2.2)$$

$$[\mathsf{and\_leaves}] \qquad \forall s_1 \in State_{\mathsf{or}}. \exists s_2 \in State_{\mathsf{and}}. s_1 \searrow s_2 \qquad (2.3)$$

$$[\mathsf{alternation}] \qquad \forall s_1, s_2 \in State. s_2 \searrow s_1 \Rightarrow s_1 \in State_{\mathsf{and}} \wedge$$

$$s_2 \in State_{\mathsf{or}} \vee (s_1 \in State_{\mathsf{or}} \wedge s_2 \in State_{\mathsf{and}}) \qquad (2.4)$$

$$[\mathsf{rooted}] \qquad \forall s \in State. root \searrow^* s \qquad (2.5)$$

$$[\mathsf{acyclic}] \qquad \forall s_1, s_2 \in State. \neg(s_2 \searrow^+ s_1 \wedge s_1 \searrow^+ s_2) \qquad (2.6)$$

$$[\mathsf{one\_parent}] \qquad \forall s_1, s_2, s_3 \in State. s_2 \searrow s_1 \wedge s_3 \searrow s_1 \Rightarrow s_2 = s_3 \qquad (2.7)$$

The substate relation imposes a tree on states, rooted in *root*. The *root* node is an or-state and all leaves are and-states. State types alternate between and and or on all paths from *root* to leaves. If $s_1 \searrow s_2$ then we say

that s_2 is a child of s_1 with $s_1 = parent(s_2)$ and $s_2 \in children(s_1)$. Members of the path from s to *root* are denoted by $ancest^*(s)$. The reflexive transitive closure of $children(s)$ is written $descend^*(s)$ and contains all descendants of s including s itself. Also and-states with no children are often called *basic* states (or-states always have children). Below we present some values for the example of Fig. 2.1:

$$State_{\mathsf{and}} = \{A, B, C_1, C_2, D_1, D_2, E_1, C_{11}, C_{12}, D_{11}, D_{12}, E_{11}, E_{12}\}$$
$$State_{\mathsf{or}} = \{root, A', D, E, C_1', D_1', E_1'\}$$
$$(\searrow) = \{(root, A), (root, B), (A, A'), (B, D), (B, E), (A', C_1),$$
$$(A', C_2), (D, D_1), (D, D_2), (E, E_1), (C_1, C_1'), (D_1, D_1'), \dots\}$$
$$parent(C_1) = A', \ children(A') = \{C_1, C_2\}, \ ancest^*(A') = \{A', A, root\},$$
$$descend^*(A') = \{A', C_1, C_2, C_1', C_{11}, C_{12}\}$$

Traditionally regions (or-states) inside non-concurrent and-states are not named in the visual syntax for statecharts. For example a child of state A has no explicit name. We have chosen to name all such states implicitly by suffixing the name of their parents with a prime symbol. For example on Fig. 2.1: $children(A) = \{A'\}$.

Definition 2.2. *The* nearest common ancestor *of a set of* and-states X, *written* $NCA(X)$, *is a state* y *such that:*

[common]	$\forall x \in X. \ y \searrow^* x$	(2.8)
[nearest]	$\forall s \in State. \ (\forall x \in X. \ s \searrow^* x) \Rightarrow s \searrow^* y$	(2.9)

For any non-empty set of and-states X, $NCA(X)$ always exists and is uniquely determined, because the $\searrow$ relation defines a single rooted tree.

Definition 2.3. *Two* and-states s_1 *and* s_2 *are orthogonal, written* $s_1 \perp s_2$ *if their NCA is an* and-state: $NCA(s_1, s_2) \in State_{\mathsf{and}}$. *A set of* and-states S *is orthogonal if all pairs of its members are orthogonal.*

In our example the set $\{D_{11}, E_1\}$ is orthogonal, while $\{D_{11}, C_2\}$ is not, since $NCA(D_{11}, C_2) = root$, which is not an and-state.

The following simple fact is useful in stating basic proofs ad absurdum:

Proposition 2.4. *Any set having a non-orthogonal subset is non-orthogonal itself.*

Each or-state s has a unique distinguished child marked ●▸, or ⒣▸. In the latter case, where the mark encloses the "H" letter, the state is called a *shallow history state* or simply a *history state*. The set of all history states is denoted *his*, so $his \subseteq State_{\mathsf{or}}$.

Whenever a history state is entered, the child that was active most recently is entered. For non-history or-states state $ini(s)$ is activated instead. On Fig. 2.1 D is the only history state, so $his = \{D\}$ and $ini = [root \mapsto A, A' \mapsto C_1, C_1' \mapsto C_{11}, D \mapsto D_1, D_1' \mapsto D_{11}, E \mapsto E_1, E_1' \mapsto E_{11}]$.

Definition 2.5. *An initial marking* $ini : State_{or} \rightarrow State_{and}$ *is a total function such that* $\forall s \in State_{or}.\ s \searrow ini(s)$. *Members of* $rng(ini)$ *are called* initial states.

2.1.2 Type System

Entities such as variables, functions and expressions are typed. Their types resemble data types of typical programing languages. Types are only moderately significant for visualSTATE code generators since all typed entities are forwarded to the underlying C compiler, which applies its standard type checking algorithm. This is why we only sketch the type system, without giving the actual typing rules. Type-correctness checks have been occasionally embedded in operational rules. This does not mean that the language is dynamically typed, but reflects our choice of not giving a full static semantics here.

We distinguish the following simple arithmetic types in visualSTATE:

$$SimpleType = \{\ [\mathsf{a..b}] \mid \mathsf{a}, \mathsf{b} \in \mathbb{Z}\ \wedge\ \mathsf{a} \leq \mathsf{b}\ \}\ \cup\ \{\mathsf{float}, \mathsf{double}\} \qquad (2.10)$$

The domain of the $[\mathsf{a..b}]$ type is the set of all integers in the $[\mathsf{a; b}]$ interval, including the endpoints. Some ranges are conveniently abbreviated as $\mathsf{int_8}, \mathsf{uint_8}, \mathsf{int_{16}}, \mathsf{uint_{16}}, \mathsf{int_{32}}, \mathsf{uint_{32}}, \mathsf{int}$, and uint. The latter two denote default integer types of the C compiler on a given platform. The actual bounds permitted for interval types are also platform dependent.

Simple types can be aggregated in vectors, so the set of all types is:

$$Type = \{\mathsf{void}\}\ \cup\ SimpleType\ \cup$$
$$\cup\ \{t[n] \mid t \in SimpleType \wedge n > 0 \wedge n \in D(\mathsf{uint})\} \quad (2.11)$$

The domain operator $D(t)$ returns the set of values of type t (a set of integers, rational numbers or respective vectors for a non-simple type). The domain of void is a singleton set containing the unit value: $D(\mathsf{void}) = \{()\}$.

Range types have been introduced in visualSTATE primarily for the sake of verification: both to reduce the domain sizes in symbolic model checking and to support detection of bounds violation. In visualSTATE no overflows are checked dynamically—the values of interval types are always represented as values of the nearest integer type and thus the standard type system of C applies at runtime.

Occasionally we will need to write conditions about types of various nontrivial elements (such as expressions). Instead of giving exact typing rules, we will use a type oracle function $\tau[\![\cdot]\!]$, when expressing these conditions.

2.1.3 Expressions, Actions and Guards

A restricted form of C arithmetic expressions is supported in the language. They are generated by the following grammar:

$$Exp \quad ::= \quad v \mid a \mid a[Exp] \mid unop\ Exp \mid Exp\ binop\ Exp \mid Aexp \quad (2.12)$$
$$Aexp \quad ::= \quad f(Exp, ..., Exp), \quad (2.13)$$

where v is a constant (integer or real in the domain of one of the supported types), a is a variable access, $a[Exp]$ is an array access, *unop* ranges over unary C operators (pure operators only, so incrementation and decrementation is not supported), *binop* ranges over binary C operators except for the assignment operators, and f ranges over the names of actions (functions). We shall also distinguish a syntactic category of assignments:

$$Assgn \quad ::= \quad v\ \texttt{"="}\ Exp \mid a[Exp]\ \texttt{"="}\ Exp\ , \quad (2.14)$$

where v ranges over scalar variable names (variables of simple types) and a ranges over array names (variables of array type).

We will say that an expression is *pure* if it does not have any side effects (this requirement includes also functions called in the expression). All expressions in the visualSTATE language are required to be pure. An expression e is *closed* in a set of variables V if all variables referred to in e are members of V. An assignment $v = e$ is closed in V if e is closed in V and $v \in V$.

Recall that *Event* denotes a finite set of events, *Signal* $\subseteq$ *Event* a finite set of internal events (signals), and *Action* a finite set of model generated outputs (actions). Also let *Var* be an abbreviation for the set of all variables: $Var = Var_I \cup Var_E$. Variables, events and actions are typed.

Definition 2.6. *Variable typing is a total function over the finite set of variables Var:* $\Gamma_V : Var \to Type$.

Definition 2.7. *Action typing is a total function:*

$$\Gamma_F : Action \to SimpleType \times SimpleType^*\ ,$$

where the first component of $\Gamma_F(f)$ *is the return type of function* f, *while the second component determines types of the parameters.*

In visualSTATE events may be parameterized with simple constant values. This allows events to carry read outs of sensors and alike. The types of the parameters are described by an event typing Γ_E:

Definition 2.8. *Event typing is a total function over the set of events:* $\Gamma_E : Event \to SimpleType^*$, *where components of* $\Gamma_E(e)$ *describe types of respective parameters of event* e.

Following are the typings for the example of Fig. 2.1:

$$Signal = \{s_1\}$$

$$\Gamma_E = [e_1 \mapsto \langle\rangle, e_2 \mapsto \langle\mathsf{int}\rangle, s_1 \mapsto \langle\rangle]$$

$$\Gamma_F = [o_1 \mapsto (\mathsf{void}, \langle\rangle), o_2 \mapsto (\mathsf{void}, \langle\rangle), enterA \mapsto (\mathsf{int}, \langle\rangle),$$
$$exitA \mapsto (\mathsf{void}, \langle\mathsf{int}\rangle), reserve \mapsto (\mathsf{void}, \langle\rangle), release \mapsto (\mathsf{void}, \langle\rangle),$$
$$lightOn \mapsto (\mathsf{void}, \langle\rangle), lightOff \mapsto (\mathsf{void}, \langle\rangle)]$$

$$\Gamma_V = [x \mapsto \mathsf{int}, y \mapsto \mathsf{float}] \tag{2.15}$$

In current implementations of both IAR visualSTATE and SCOPE parameterized signals (i.e. *internal* events) are not supported, so $\forall e \in Signal. \Gamma_E(e) = \langle\rangle$, providing for space-efficient implementations of signal queue.

As the type $\Gamma_E(e)$ describes types of parameters of event e, a *binding* $e(p_1, \ldots, p_k)$ assigns local names to these parameters.

Definition 2.9. *A name binding for an event e is a term $e(p_1, ..., p_k)$, where e is an event and $p_1, ..., p_k$ are variable names, i.e.*

$$Ebind = \{e(p_1, ..., p_k) \mid k = |\Gamma_E(e)| \ \wedge \ \forall i \in \{1..k\}. \ p_i \in Var\}$$

The identifiers $p_1, ..., p_k$ will usually be fresh in the context. If a name of an existing variable is used, then the parameter will hide the existing variable in the scope of the event binding (the scope extends over a single transition).

We will mix assignments, action expressions and signals in sequences of type $(Aexp|Assgn|Signal)^*$. Such sequences are embedded into each transition and each and-state. Actions embedded in states play role similar to actions of Moore machines [96]. One such action is performed whenever the state is entered (*entry action*). The *exit action* is executed whenever the state is being exited. Two action mappings for states determine what actions are executed for what states:

$$en : State_{\mathsf{and}} \to (Aexp|Assgn|Signal)^*$$
$$ex : State_{\mathsf{and}} \to (Aexp|Assgn|Signal)^* \tag{2.16}$$

Note that these mappings are total. States that do not have actions assigned in visual syntax, are assigned an empty list of actions $\langle\rangle$ in the abstract syntax. In Fig. 2.1:

$$en = [\mathsf{A} \mapsto \langle x = enterA()\rangle, \mathsf{B} \mapsto \langle\rangle, \mathsf{C}_1 \mapsto \langle reserve()\rangle,$$
$$\mathsf{C}_2 \mapsto \langle\rangle, \mathsf{D}_1 \mapsto \langle\rangle, \mathsf{D}_2 \mapsto \langle\rangle, \mathsf{E}_1 \mapsto \langle lightOn()\rangle, \ldots]$$
$$ex = [\mathsf{A} \mapsto \langle exitA(x+1)\rangle, \mathsf{B} \mapsto \langle\rangle, \mathsf{C}_1 \mapsto \langle release()\rangle,$$
$$\mathsf{C}_2 \mapsto \langle\rangle, \mathsf{D}_1 \mapsto \langle\rangle, \mathsf{D}_2 \mapsto \langle\rangle, \mathsf{E}_1 \mapsto \langle lightOff()\rangle, \ldots] \tag{2.17}$$

While expressions are evaluated over implicit state (event parameters and variables), guards are evaluated over explicit state (state configurations). They are generated according to the following grammar:

$$Guard ::= true \mid s \mid \neg s \mid Guard \wedge Guard \;, \tag{2.18}$$

where s ranges over and-states. Both the separation of guards from expressions and the syntax of guards (the lack of disjunction) are IAR visualSTATE specific. We have chosen to adhere to this strict syntax for two reasons. First, to make comparisons against the industrial toolkit more easy and direct. Second, we believe that such rigid syntactic rules enforce a safer[1] and resource aware modeling style, which is desirable in the embedded domain.

Note that, as we shall see later in the evaluation rules, the value of guards is always deterministic as guards are evaluated over the current state configuration. Possible state changes due to transition firing, do not affect values of guards, as these only change activity of states in the *next* configuration. This feature dates back to original Harel statecharts.

2.1.4   Transitions

We distinguish syntactic and semantic transitions. The former are explicitly drawn in the model and connect states. Semantic transitions collect syntactic transitions in sets fired in a single step moving from a set of state to another set of states. In this sense semantic transitions are more abstract. They belong to the labeled transition system, which is the execution model of given statechart.

Each system has a finite set of syntactic transitions *Trans* such that:

$$\begin{aligned} Trans \;\subseteq\;\; & Ebind \times State_{\mathsf{and}} \times Guard \times Exp \times \\ & \times (Aexp|Assgn|Signal)^* \times \mathcal{P}(State_{\mathsf{and}}) \end{aligned}$$

The components of a transition t are (from left to right): a triggering event $event(t)$ with parameter names $params(t)$, the source and-state $source(t)$, the guard $guard(t)$, a pure conditional expression $expr(t)$, a sequence of actions and a non-empty orthogonal set of target and-states $targets(t)$.

All expressions and assignments on the transition $t = (E(p_1, \ldots, p_k), s_0, g, e, as, \{s_1, \ldots, s_n\})$ must be closed in $Var \cup \{p_1, \ldots, p_k\}$. For clarity of further presentation we assume that the source state is already included in the guard condition (so $g \Rightarrow s_0$).

---

[1]It is feasible to exactly interpret guards over explicit state when doing model checking. Treatment of expressions is more expensive and often has to be based on imprecise overapproximations, due to external function calls and externally accessible variables.

Following are the syntactic transitions of Fig. 2.1:

$$t_1 = (e_1(), \mathsf{D}_1, \mathsf{D}_1 \wedge \mathsf{D}_{12} \wedge \neg \mathsf{E}_{11}, 1, \langle o_1(), s_1 \rangle, \{\mathsf{C}_{11}\})$$
$$t_2 = (s_1, \mathsf{A}, \mathsf{A} \wedge \neg \mathsf{C}_2, 1, \langle o_2() \rangle, \{\mathsf{B}\})$$
$$t_3 = (e_2(n), \mathsf{C}_2, \mathsf{C}_2, x == 1, \langle\rangle, \{\mathsf{D}_{12}, \mathsf{E}_1\})$$
$$t_4 = (e_2(n), \mathsf{C}_1, \mathsf{C}_1, 1, \langle x = (x+n)\%2 \rangle, \{\mathsf{C}_1\}) \tag{2.19}$$

Often we will write the transitions in more intuitive way using an arrow, dropping superflous empty parentheses and explicit source in the guard. For example t_1 could be written as:

$$\mathsf{D}_1 \xrightarrow{e_1 \; [(\mathsf{D}_{12} \wedge \neg \mathsf{E}_{11}) \; (1)]/\langle o_1, s_1 \rangle} \{\mathsf{C}_{11}\} \tag{2.20}$$

2.2 Dynamic Semantics

IAR visualSTATE statecharts are reactive synchronous systems. Reactive means that the system continuously responds to a stream of incoming events, and synchronous means that the reaction to each event can be considered infinitely fast [10, 55]. In practice this means that the system must be much faster than the environment. Full synchrony in the sense of Berry [10] is not guaranteed, though, due to the explicit microsteps and the use of queues.

We define the dynamic semantics of a IAR visualSTATE system by describing how expressions are evaluated, reactions executed, states (scopes) exited and entered, and transitions fired. Then we describe the processing of a single internal event in a so-called microstep comprising all transitions fired in response to a single event, and specify how a sequence of microsteps makes up a macrostep, processing a single external event. A single microstep can place internal events in an internal signal queue. This internal events have to be served one-by-one in the very same way as the external events are. Thus microsteps are itereated until the queue is discharged. This complete iteration is called a macrostep.

Last but not least we also say how a system is initialized.

2.2.1 Runtime State and Values

A variable store gives the values of system variables at a given point of execution. Values of variables can be integer and floating point numbers, and vectors thereof.

Definition 2.10. *A store ϱ is a total mapping of type*

$$Store \subseteq Var \to Value, \quad where \quad Value = \bigcup_{t \in Type} D(t)$$

and $\forall x \in Var. \; \varrho(x) \in D(\Gamma_V(x))$.

Event instances describe the actual values of event parameters at run-time.

Definition 2.11. *An instance of an event e is a term $e(v_1, ..., v_k)$, where $v_1, ..., v_k$ are values consistent with event typing, i.e.*

$$Einst = \{ \ e(v_1, ..., v_k) \mid k = |\Gamma_E(e)| \wedge \forall i \in \{1..k\}.v_i \in D(\pi_i(\Gamma_E(e))) \ \} \ .$$

Similarly action instances are function prototypes where formal parameters have been substituted with the actual values:

Definition 2.12. *An Ainst is a term $f(v_1, ..., v_k)$, where $v_1, ..., v_k$ are constants:*

$$Ainst = \{f(v_1, ..., v_k) \mid \forall i \in \{1..k\}. \ v_i \in D(\pi_i(\pi_2(\Gamma_F(f)))) \ \}$$

A signal queue is a list of pending local events, i.e. events that have been signaled as result of some actions and have not yet been processed.

Definition 2.13. *A signal queue is an ordered list of signals and event instances: $q \in Queue = (Signal|Einst^*)$.*

The signal queue is a FIFO queue. The notational convention is that new elements are concatenated at the right end of the list (suffixed). In this sense the caret symbol ˆ is a queue constructor. Concatenation in front (left end) is used to express internal structure of the list (prefixed event instance). It should be understood as pattern matching.

Definition 2.14. *A set of states X is a* maximal orthogonal set *of substates of s iff adding any new descendant of s to X would create a non-orthogonal set, i.e. $\forall y \in State \setminus X. \ s \searrow^* y \Rightarrow X \cup \{y\}$ is non-orthogonal.*
A maximal orthogonal set of substates of s is called a configuration *of s if it contains only basic states.*

A configuration of *root* state is called a global state configuration of the statechart. The initial configuration is uniquely determined by the function *ini*. The set of all configurations is denoted Σ, the set of all global configuration is written Σ_{root}, and the set of all maximal orthogonal subsets of *State* is written Σ_{max}.

The set $\{D_{11}, E_1\}$ is a maximal orthogonal set. It is not a configuration though. $\{D_{11}, E_{12}\}$ is a global state configuration (a configuration of *root*).

A *history marking* maintains runtime information of previously active children of history states: for each history state (an or-state in the *his* set), it memoizes which of its children was active most recently.

Definition 2.15. *A history marking η is a total function of type $his \rightarrow State_{and}$ such that $\forall s \in dom(\eta). \ s \searrow his(s)$.*

The initial history marking η_0 is defined as a restriction of initial marking: $\eta_0 = ini|_{his}$.

Definition 2.16. *The state of the system $(\sigma, \varrho, \eta, q)$ consists of the current state configuration σ, current store value ϱ, current history marking η, and current signal queue q.*

2.2.2 Expression Evaluation

Function calls and expressions are evaluated in the current variable store. We will avoid giving the detailed expression semantics here and denote it by an overloaded symbol $\xrightarrow[\text{C-sem}]{}$:

$$\xrightarrow[\text{C-sem}]{} \subseteq Store \times Exp \times Value \tag{2.21}$$

$\langle \varrho, e \rangle \xrightarrow[\text{C-sem}]{} \langle v \rangle$, where $e \in Exp$ and $v \in D(\tau[\![e]\!])$ is the result of evaluation

of e in the C semantics. $\tag{2.22}$

$$\xrightarrow[\text{C-sem}]{} \subseteq Store \times Ainst \times Value \times Store$$

$\langle \varrho_0, f(v_1, \ldots, v_k) \rangle \xrightarrow[\text{C-sem}]{} \langle v, \varrho_1 \rangle$, where $f(v_1, \ldots, v_k) \in Ainst$ and $v \in D(\pi_1(\Gamma_F(f)))$

is the value returned in the C semantics. $\tag{2.23}$

Note that the first rule (2.22) does not modify the existing store. This is because our expressions are pure. Functions may be impure when used outside expressions (as actions). For pure functions we have $\varrho_0 = \varrho_1$ in rule 2.23. Calls to impure functions may modify values of externally accessible variables, but values of internal variables stay intact:

$$\varrho_0\big|_{Var_I} = \varrho_1\big|_{Var_I} \tag{2.24}$$

Following are the rules for executing assignments. As our expressions are required to be pure, the only side effect of the assignment is the change of the modified variable itself. Type coercions are not shown explicitly, to avoid clutter, but types are promoted according to ISO C rules [59, section 6.3].

$$\xrightarrow[\text{asgn}]{} \subseteq Assgn \times Store \times Store \tag{2.25}$$

$$\frac{\langle e, \varrho \rangle \xrightarrow[\text{C-sem}]{} \langle v \rangle}{\langle x = e, \varrho \rangle \xrightarrow[\text{asgn}]{} \varrho[v/x]} \tag{2.26}$$

$$\frac{\langle e_0, \varrho \rangle \xrightarrow[\text{C-sem}]{} \langle v_0 \rangle \quad v_0 \geq 0 \quad \langle e_1, \varrho \rangle \xrightarrow[\text{C-sem}]{} \langle v_1 \rangle \quad v = \varrho(x)[v_1/v_0]}{\langle x[e_0] = e_1, \varrho \rangle \xrightarrow[\text{asgn}]{} \varrho[v/x]} \tag{2.27}$$

An output relation specifies the externally visible changes to the environment. These changes are performed by means of standalone function calls placed on transitions and as parts of entry and exit actions in states. In order to emphasize that the labels of the action relation are visible externally we will suffix them by the exclamation mark.

$$\xrightarrow[\text{output}]{} \subseteq Aexp \times Store \times Ainst \times Store \tag{2.28}$$

$$\frac{\forall i \in \{1..k\}.\langle e_i, \varrho_0 \rangle \xrightarrow[\text{C-sem}]{} \langle v_i \rangle \quad \langle f(v_1, \ldots, v_k), \varrho_0 \rangle \xrightarrow[\text{C-sem}]{} \langle \_, \varrho_1 \rangle \quad \varrho_0 \big|_{Var_I} = \varrho_1 \big|_{Var_I}}{\langle f(e_1, \ldots, e_k), \varrho_0 \rangle \xrightarrow[\text{output}]{f(v_1, \ldots, v_k)!} \langle \varrho_1 \rangle}$$

$$\tag{2.29}$$

Note that in the above rule f is not required to be pure. On the contrary, it is supposed to have an effect on the environment of the program, and perhaps on some variables in store.

2.2.3   Parameterizing the Semantics

Actions in a list are executed consecutively. We first give a variant of the rules that precisely reports in what order actions are executed (sequence-based outputs). Such level of granularity is needed if the side effects of function calls are interdependent.

$$\xrightarrow[\text{exec}]{} \subseteq (Assgn|Aexp|Signal)^* \times Store \times Queue \times Ainst^* \times Store \times Queue$$

$$\tag{2.30}$$
$$\frac{}{\langle \langle \rangle, \varrho, q \rangle \xrightarrow[\text{exec}]{\langle \rangle!}{}_* \langle \varrho, q \rangle}$$

$$\frac{a \in Assgn \quad \langle a, \varrho_0 \rangle \xrightarrow[\text{asgn}]{} \langle \varrho_1 \rangle \quad \langle tl, \varrho_1, q_0 \rangle \xrightarrow[\text{exec}]{os!}{}_* \langle \varrho_2, q_1 \rangle}{\langle \langle a \rangle\hat{\ }tl, \varrho_0, q_0 \rangle \xrightarrow[\text{exec}]{os!}{}_* \langle \varrho_2, q_1 \rangle} \tag{2.31}$$

$$\frac{a \in Aexp \quad \langle a, \varrho_0 \rangle \xrightarrow[\text{output}]{o!} \langle \varrho_1 \rangle \quad \langle tl, \varrho_1, q_0 \rangle \xrightarrow[\text{exec}]{os!}{}_* \langle \varrho_2, q_1 \rangle}{\langle \langle a \rangle\hat{\ }tl, \varrho_0, q_0 \rangle \xrightarrow[\text{exec}]{(\langle o \rangle\hat{\ }os)!}{}_* \langle \varrho_2, q_1 \rangle} \tag{2.32}$$

$$\frac{s \in Signal \quad \langle tl, \varrho_0, q_0\hat{\ }\langle s \rangle \rangle \xrightarrow[\text{exec}]{os!}{}_* \langle \varrho_1, q_1 \rangle}{\langle \langle s \rangle\hat{\ }tl, \varrho_0, q_0 \rangle \xrightarrow[\text{exec}]{os!}{}_* \langle \varrho_1, q_1 \rangle} \tag{2.33}$$

An alternative variant of the above rules is closer to the original formulation of Harel [42, 47]. It produces sets of outputs, instead of sequences, which

reflects the environment's inability to observe the order in which the side effects happen. Such a formulation is sometimes considered more synchronous. The following rules replace rules (2.30)–(2.33). The main differences are in the first (2.34) and the third rule (2.36). Observe the new type of the execution relation: $\mathcal{P}(Ainst)$ replaces $Ainst^*$. Also os denotes a set of outputs in the rules below, and we use set operations and not list constructors, in the transition labels.

$$\xrightarrow[\text{exec}]{}_{\mathcal{P}} \subseteq (Assgn|Aexp|Signal)^* \times Store \times Queue \times \mathcal{P}(Ainst) \times Store \times Queue$$

$$\frac{}{\langle\langle\rangle, \varrho, q\rangle \xrightarrow[\text{exec}]{\emptyset!}_{\mathcal{P}} \langle\varrho, q\rangle} \tag{2.34}$$

$$\frac{a \in Assgn \quad \langle a, \varrho_0\rangle \xrightarrow[\text{asgn}]{} \langle\varrho_1\rangle \quad \langle tl, \varrho_1, q_0\rangle \xrightarrow[\text{exec}]{os!}_{\mathcal{P}} \langle\varrho_2, q_1\rangle}{\langle\langle a\rangle\hat{\ }tl, \varrho_0, q_0\rangle \xrightarrow[\text{exec}]{os!}_{\mathcal{P}} \langle\varrho_2, q_1\rangle} \tag{2.35}$$

$$\frac{a \in Aexp \quad \langle a, \varrho_0\rangle \xrightarrow[\text{output}]{o!} \langle\varrho_1\rangle \quad \langle tl, \varrho_1, q_0\rangle \xrightarrow[\text{exec}]{os!}_{\mathcal{P}} \langle\varrho_2, q_1\rangle}{\langle\langle a\rangle\hat{\ }tl, \varrho_0, q_0\rangle \xrightarrow[\text{exec}]{(\{o\}\cup os)!}_{\mathcal{P}} \langle\varrho_2, q_1\rangle} \tag{2.36}$$

$$\frac{s \in Signal \quad \langle tl, \varrho_0, q_0\hat{\ }\langle s\rangle\rangle \xrightarrow[\text{exec}]{os!}_{\mathcal{P}} \langle\varrho_1, q_1\rangle}{\langle\langle s\rangle\hat{\ }tl, \varrho_0, q_0\rangle \xrightarrow[\text{exec}]{os!}_{\mathcal{P}} \langle\varrho_1, q_1\rangle} \tag{2.37}$$

We could produce more output variants in a very similar way. Unfortunately we would always have to rewrite the above four rules, and many other rules to follow. Ceasing to be so explicit, we shall parameterize our semantics with the type of output structure, and introduce abstract constructors for outputs that can be instantiated for lists, sets and other structures.

Each output structure shall be characterized using four values: the type of outputs denoted $[Ainst]$, the empty output constructor denoted $\bot$, the *cons* constructor for adding a single action instance to the produced output, and $\prod$ constructor for composing several (fragments of) outputs into a single output. The types for the parameters are:

$$\bot : [Ainst], \ cons : Ainst \times [Ainst] \to [Ainst], \ \prod : [[Ainst]] \to [Ainst] \tag{2.38}$$

We will also use a derived constructor $cons^*$—a folding of *cons* over one

of the parameters:

$$cons^* : [Ainst] \times [Ainst] \to [Ainst]$$
$$cons^*(\bot, ys) = ys$$
$$cons^*(cons(x, xs), ys) = cons(x, cons^*(xs, ys)) \qquad (2.39)$$

Note that $cons^*$ is very similar to $\prod$. Indeed they will be the same for some typical output structures, but for some they will differ. Most notably $cons^*$ is meant to be order preserving, while $\prod$ is not necessarily (indeed we have not put any restrictions on $\prod$).

We shall consistently use these three parameters, instead of concrete constructors, in the semantics rules. For example rules (2.32,2.36) can be rewritten:

$$\frac{a \in Aexp \quad \langle a, \varrho_0 \rangle \xrightarrow[\text{output}]{o!} \langle \varrho_1 \rangle \quad \langle tl, \varrho_1, q_0 \rangle \xrightarrow[\text{exec}]{os!} \langle \varrho_2, q_1 \rangle}{\langle \langle a \rangle \hat{\ } tl, \varrho_0, q_0 \rangle \xrightarrow[\text{exec}]{cons(o, os)!} \langle \varrho_2, q_1 \rangle} \qquad (2.40)$$

Table 2.1 summarizes interpretations for four most popular output structures in statecharts semantics: sets, multisets (bags), sequences and interleavings of sequences. The first row presents a plain set-based output structure as used in rules (2.34)–(2.37). This kind of outputs is typically associated with modeling hardware, where outputs correspond to signal wires. It was used in early semantics of statecharts, for example [47]. A slight extension of this semantics, the second row in the table, assumes outputs to be multisets of actions. In this variant, among the others used in [5], the environment is able to observe how many times a given output was produced during a single reaction. Sequence based semantics, the third row, is prevailing in code generators, as they usually synthesize deterministic sequential programs [57, 137][103, chap.15]. As we have mentioned introducing rules (2.30)–(2.33), this variant of the semantics assumes an interdependence between side effects, or in other words—the ability of environment to observe the exact order in which the outputs are produced (in contrast to mere observation of the kind of outputs, or numbers of occurrences of outputs produced).

The last entry in Table 2.1 is a nondeterministic generalization of the previous row. Despite the fact that synthesized programs are deterministic, code generators often internally use somewhat nondeterministic semantics. This reflects underspecifications like unknown order of processing of transitions and a concurrent (interleaving) execution of transitions. The correctness of code generation is based on a refinement relation, which amounts to picking-up some legal sequentialization of the abstract output. The interleaving of two sequences X and Y, written $X \parallel Y$, is understood as a

output structure		$[X]$	$\bot$	$cons : [X] \times X \to [X]$	$\prod : [[X]] \to [X]$
set	$\to_{\mathcal{P}}$	$\mathcal{P}(X)$	$\emptyset$	$\lambda(x,xs).\{x\} \cup xs$	$\lambda\mathbb{X}.\bigcup\mathbb{X}$
multiset	$\to_{\mathcal{M}}$	$\mathcal{M}(X)$	$\emptyset$	$\lambda(x,xs).\{x\} \uplus xs$	$\lambda\mathbb{X}.\biguplus\mathbb{X}$
sequence	$\to_*$	X^*	$\langle\rangle$	$\lambda(x,xs).\langle x\rangle\hat{\ }xs$	$\lambda\langle X_1,\ldots,X_n\rangle.X_1\hat{\ }\ldots\hat{\ }X_n$
interleaved seq.	$\to_{**}$	X^*	$\langle\rangle$	$\lambda(x,xs).\langle x\rangle\hat{\ }xs$	$\lambda\langle X_1,\ldots,X_n\rangle.X_1\parallel\ldots\parallel X_n$

Table 2.1: Output structure interpretations for the most typical variants of statecharts.

sequence of elements of X and Y interleaved, i.e. mixed in a way that relative orderings of elements within original sequences are preserved. Note that this operator is nondeterministic.

As we shall soon see, the semantics of statecharts allows some freedom in choosing the order of processing states. We use a partial order on states $\blacktriangleleft\subseteq State \times State$ to model permissible choices in the ordering over states. If the order in which concurrent processing should traverse the hierarchy is entirely nondeterministic then $\blacktriangleleft$ is empty. In many implementations this priority will amount to some known traversal of the hierarchy tree (pre/in/post-order).

The impact of the state priority ordering on the observable behavior is limited by the actual output structure (table 2.1). For example processing order for states is irrelevant if the outputs are modeled as sets. Regardless of what processing order is used the same set of actions will be generated. On the other extreme, priority controls very fine grained subtleties in the semantics, if the outputs are modeled as sequences of actions. In such case the order of every processing is highly meaningful and leads to a different observable behavior.

2.2.4   Exiting and Entering States

We shall now begin to describe the dynamics of firing transitions. A firing of single transition consists of deciding whether it is enabled, computing its scope, exiting the scope and entering the targets. The *scope* is a transition specific or-state that describes an area of impact of this transitions. A given transition only changes the current state configuration within its scope. We shall formalize the notion of scope and various aspects of transition firing in later sections. Presently we are interested in discussing the mechanism of exiting an or-state (that will soon prove to be scope, so we dare to use this name already now) and entering target states.

As we have said before, each state has an exit action assigned. This action is executed whenever the state is exited, entailing execution of the exit actions of all descendant states (bottom-up). Exiting affects the entire hierarchy below a given or-state, a *scope*. The scope itself is not left but

all its active descendants are.   Following rules describe exiting for a basic and-state, a nonhistory and history or-states, and a non-basic and-state respectively:

$$\overrightarrow{\text{exit}} \subseteq State \times \Sigma_{\max} \times Store \times History \times Queue \times$$
$$\times [Ainst] \times Store \times History \times Queue \qquad (2.41)$$

$$\frac{s \in State_{\text{and}} \quad children(s) = \emptyset \quad \sigma = \{s\}}{\langle s, \sigma, \varrho, \eta, q \rangle \xrightarrow[\text{exit}]{\perp!} \langle \varrho, \eta, q \rangle} \qquad (2.42)$$

$$\frac{\begin{array}{c} s \in State_{\text{or}} \ \wedge\ s \searrow s' \ \wedge\ \sigma \subseteq descend^*(s') \ \wedge\ s \notin dom(\eta_0) \\ \langle s', \sigma, \varrho_0, \eta_0, q_0 \rangle \xrightarrow[\text{exit}]{os_0!} \langle \varrho_1, \eta_1, q_1 \rangle \ \wedge\ \langle \langle ex(s'), \varrho_1, q_1 \rangle \rangle \xrightarrow[\text{exec}]{os_1} \langle \varrho_2, q_2 \rangle \end{array}}{\langle s, \sigma, \varrho_0, \eta_0, q_0 \rangle \xrightarrow[\text{exit}]{cons^*(os_0, os_1)!} \langle \varrho_2, \eta_1, q_2 \rangle} \qquad (2.43)$$

$$\frac{\begin{array}{c} s \in State_{\text{or}} \ \wedge\ s \searrow s' \ \wedge\ \sigma \subseteq descend^*(s') \ \wedge\ s \in dom(\eta_0) \\ \langle s', \sigma, \varrho_0, \eta_0, q_0 \rangle \xrightarrow[\text{exit}]{os_0!} \langle \varrho_1, \eta_1, q_1 \rangle \ \wedge\ \langle ex(s'), \varrho_1, q_1 \rangle \xrightarrow[\text{exec}]{os_1!} \langle \varrho_2, q_2 \rangle \end{array}}{\langle s, \sigma, \varrho_0, \eta_0, q_0 \rangle \xrightarrow[\text{exit}]{cons^*(os_1, os_2)!} \langle \varrho_2, \eta_1[s'/s], q_2 \rangle} \qquad (2.44)$$

$$\frac{\begin{array}{c} s \in State_{\text{and}} \ \wedge\ \{s_1, ..., s_k\} = children(s) \ \wedge\ \forall i, j \in \{1..k\}.\ i < j \Rightarrow \neg(s_j \blacktriangleleft s_i) \\ \forall i \in \{1..k\}.\ \langle s_i, \sigma \cap descend^*(s_i), \varrho_{i-1}, \eta_{i-1}, q_{i-1} \rangle \xrightarrow[\text{exit}]{os_{i-1}!} \langle \varrho_i, \eta_i, q_i \rangle \end{array}}{\langle s, \sigma, \varrho_0, \eta_0, q_0 \rangle \xrightarrow[\text{exit}]{\prod(os_0, ..., os_{k-1})!} \langle \varrho_k, \eta_k, q_k \rangle}$$
$$(2.45)$$

The exit rules are executed in a bottom-up order. First the most nested descendants are exited, then their parents and so on recursively until the direct children of s. Descendants of and-state components are exited in the components' priority ordering $\blacktriangleleft$. If $\blacktriangleleft$ orders children from left to right, then the exit relation performs a postorder traversal of the statechart hierarchy. If for every and-state s the priority $\blacktriangleleft$ is a total order on its *children*(s), then the order of the exiting is deterministic. Otherwise it is non-deterministic. Also note that the history marking η (see definition 2.15) is updated directly after a state has been exited (rule 2.44).

The well-formedness of the exit rules (that they are always called on a proper state configuration) follows from proposition 2.17:

Proposition 2.17. *Let σ be a state configuration of s, and s' a child of state s ($s \searrow s'$). Then $\sigma' = \sigma \cap descend^*(s')$ is itself a state configuration of s'.*

Proof. Orthogonality follows from the fact that any subset of the orthogonal set is itself orthogonal. Maximality is easily shown via contrapositive. If $\sigma \cap descend^*(s')$ is not maximal then σ could not be maximal either. □

Entering a state resembles exiting. The entry rules of a state and its descendants should be executed in a proper top-down order. Moreover, for each or-state it should be determined which of its children is the default state, determined by the current history marking or the initial marking.

Definition 2.18. *The default child of a given* or-*state s in the current history marking η is given by:*

$$default(s, \eta) = \begin{cases} ini(s) & \text{if } s \notin dom(\eta) \\ \eta(s) & \text{if } s \in dom(\eta) \end{cases}$$

States are normally entered after a certain scope has been exited. So we start not with a proper configuration, but with a maximal orthogonal set of *root* containing some non-basic states. The entry path is not only indicated by initial and history markings, but also by targets indicated on the transition (in fact transition targets take precedence over the default path). Obviously the descendants of target states should also be entered in the process, via the default path. All this makes entering slightly more complex than exiting. Relation $\xrightarrow[enter]{}$ relates the set of targets T, root of the part of the hierarchy being exited s, variable store ϱ, history marking η and signal queue q, with a new state configuration of s, new store value and a new signal queue. The middle argument indicates the compounds output produced in a given enter sequence:

$$\xrightarrow[\text{enter}]{} \subseteq \mathcal{P}(State_{\mathsf{and}}) \times State \times Store \times History \times Queue \times$$

$$\times [A\,inst] \times \Sigma \times Store \times Queue \qquad (2.46)$$

$$\frac{T = \emptyset \qquad s \in State_{\mathsf{and}} \qquad children(s) = \emptyset}{\langle T, s, \varrho, \eta, q \rangle \xrightarrow[\text{enter}]{\bot!} \langle \{s\}, \varrho, q \rangle} \qquad (2.47)$$

$$\frac{\begin{array}{c} T \subseteq descend^*(s) \ \land \ s \in State_{\mathsf{and}} \\ \{s_1, ..., s_k\} = children(s) \ \land \ \forall i, j \in \{1..k\}. \ i < j \Rightarrow \neg(s_j \blacktriangleleft s_i) \\ \forall i \in \{1..k\}. \ \langle T \cap descend^*(s_i), s_i, \varrho_{i-1}, \eta, q_{i-1} \rangle \xrightarrow[\text{enter}]{os_{i-1}!} \langle \sigma_i, \varrho_i, q_i \rangle \end{array}}{\langle T, s, \varrho_0, \eta, q_0 \rangle \xrightarrow[\text{enter}]{\prod(os_0, ..., os_{k-1})!} \langle \bigcup_{i=1}^k \sigma_i, \varrho_k, q_k \rangle} \qquad (2.48)$$

$$\frac{\begin{array}{c} T \neq \emptyset \ \land \ T \subseteq descend^+(s) \ \land \ s \in State_{\mathsf{or}} \ \land \ s \searrow s' \searrow^* NCA(T) \\ \langle \langle en(s'), \varrho_0, q_0 \rangle \rangle \xrightarrow[\text{exec}]{os_0!} \langle \varrho_1, q_1 \rangle \ \land \ \langle T \setminus \{s'\}, s', \varrho_1, \eta, q_1 \rangle \xrightarrow[\text{enter}]{os_1!} \langle \sigma, \varrho_2, q_2 \rangle \end{array}}{\langle T, s, \varrho_0, his, q_0 \rangle \xrightarrow[\text{enter}]{cons^*(os_0, os_1)!} \langle \sigma, \varrho_2, q_2 \rangle} \qquad (2.49)$$

$$\frac{\begin{array}{c} T = \emptyset \ \land \ s \in State_{\mathsf{or}} \ \land \ s' = default(s, \eta) \\ \langle \langle en(s'), \varrho_0, q_0 \rangle \rangle \xrightarrow[\text{exec}]{os_0!} \langle \varrho_1, q_1 \rangle \qquad \langle \emptyset, s', \varrho_1, \eta, q_1 \rangle \xrightarrow[\text{enter}]{os_1!} \langle \sigma, \varrho_2, q_2 \rangle \end{array}}{\langle T, s, \varrho_0, \eta, q_0 \rangle \xrightarrow[\text{enter}]{cons^*(os_0, os_1)!} \langle \sigma, \varrho_2, q_2 \rangle} \qquad (2.50)$$

Well-formedness of the configuration resulting in rule 2.48 is ensured by proposition 2.19:

Proposition 2.19. *Let s be an* and*-state, $s_1, ..., s_k$ be children of s and $\sigma_1, ..., \sigma_k$ be state configurations of $s_1, ..., s_k$ respectively. Then $\sigma = \bigcup_{i=1}^k \sigma_i$ is a state configuration of s.*

Proof. Show that elements of σ are basic (trivially as elements of all σ_i are basic), that σ is an orthogonal set (any two states belonging to it either belong to the same σ_i, and are orthogonal by assumption, or belong to two different subtrees and are orthogonal by definition as s is their *NCA*), and that σ is maximal (adding any fresh basic state would make one of σ_i nonorthogonal and then by proposition 2.4 it would make the entire union nonorthogonal). □

Similarly to the exit semantics, the entry is nondeterministic, unless for every and-state s the priority order $\blacktriangleleft$ restricted to *children(s)* is a total ordering. If priority function $\blacktriangleleft$ orders children of and-states from left to right then the enter relation performs preorder traversal of statechart hierarchy. The history marking remains unchanged in the entering phase.

2.2.5 Satisfaction of Guards and Expressions

The evaluation relation induces a satisfaction relation for expressions. An expression is satisfied if it evaluates to a non-zero value:

$$\models \ \subseteq \ Store \times Exp \tag{2.51}$$

$$\varrho \models e \qquad\qquad \text{iff } \langle \varrho, e \rangle \xrightarrow[\text{C-sem}]{} \langle v \rangle \wedge v \neq 0 \tag{2.52}$$

As expressions are evaluated against the store, guards (see p. 16) are evaluated against the state configuration:

$$\models \ \subseteq \ \Sigma_{root} \times Guard \tag{2.53}$$

$$\sigma \models true \qquad\qquad \text{always} \tag{2.54}$$
$$\sigma \models s \qquad\qquad \text{iff } s \in \sigma \tag{2.55}$$
$$\sigma \models \neg s \qquad\qquad \text{iff } s \notin \sigma \tag{2.56}$$
$$\sigma \models g_0 \wedge g_1 \qquad\qquad \text{iff } \sigma \models g_0 \text{ and } \sigma \models g_1 \tag{2.57}$$

2.2.6 Firing Transitions

The scope of changes involved in switching to an arbitrary state s depends on the target state s itself and the current configuration σ. For the given target s and a configuration σ a scope state s' is found, i.e. the lowest (possibly innermost) or-state such that $s' \searrow^{*} s$ and some of its descendants are members in σ. The intuition is that the state change required to make s active should be minimal.

   This contrasts with a more standard choice of UML and Harel's semantics, where the scope of change is computed for all target states of a transition *collectively*, instead of *individually* for each of the targets [103, sec.15.3.13, p.501]. The individual semantics of scopes in visualSTATE allows incorporating changes to orthogonal regions in a single transition. In UML multiple targets cannot be used to achieve this effect, highly desired by engineers. A heavier modeling construct, signals, needs to be applied instead.

   Figure 2.2 presents a simple example demonstrating the difference between the individual scope semantics and the collective scope semantics. Assume that $\sigma = \{D, H\}$ is the current state configuration and that event e_1 arrives, causing the left-most transition of the model to fire. In the collective scope semantics (UML/Harel, Fig. 2.2, right) this transition has only one scope: the *root* state. When it fires it has to execute the exit actions of
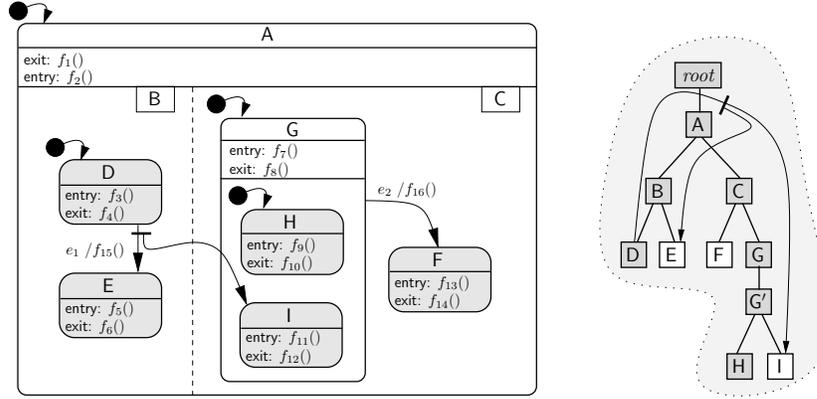
Figure 2.2: A multiple target transition. Left: syntax Right: sketch of firing semantics according to UML definition.
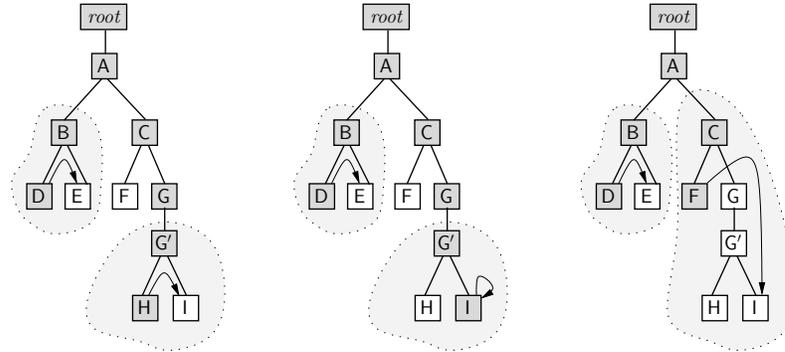
Figure 2.3: IAR visualSTATE's individual scope semantics: scopes for three different active configurations.

all active states, produce f_{15} and entry actions of all states in the configuration. This may give rise to the following sequence of actions:

$$ex(D)\,\hat{}\,ex(H)\,\hat{}\,ex(G)\,\hat{}\,ex(A)\,\hat{}\,\langle a\rangle\,\hat{}\,en(A)\,\hat{}\,en(E)\,\hat{}\,en(G)\,\hat{}\,en(I) \qquad (2.58)$$

In the individual scope semantics (visualSTATE, Fig. 2.3) the state changes are taken as locally as possible. The following sequence is one of the possible outputs (it was not legal under the collective semantics):

$$ex(D)\,\hat{}\,ex(H)\,\hat{}\,\langle a\rangle\,\hat{}\,en(E)\,\hat{}\,en(I) \qquad (2.59)$$

For a given target state s we need to find the closest relative in the current configuration (the state $s' \in \sigma$ minimizing $NCA(parent(s), parent(s'))$). The nearest common ancestor found is the (implicit) scope of the transition.

Definition 2.20. *Let s be the target state and σ the current state configuration (global state). The implicit scope of s in σ is defined recursively:*

$$iscope(\sigma, s) = \begin{cases} parent(s) & \textit{if } descend^*(parent(s)) \cap \sigma \neq \emptyset \\ iscope(\sigma, parent(s)) & \textit{otherwise} \end{cases}$$

The implicit scope determines the exit/entry impact of an actual state change. All descendants of the scope will be exited when firing a transition, and some new states (depending on targets and markings) will be entered. Each transition has several implicit scopes: at most as many as there are target states. Two orthogonal targets may have the same implicit scope (for instance when simultaneously entering several components of the same and-state).

Definition 2.21. *The generalized scope of transition t in configuration σ is the set of its implicit scopes:*

$$scope(\sigma, t) = \{ iscope(\sigma, s) \mid s \in targets'(t) \} \ ,$$

where $targets'(t)$ is a normalized set of targets of t, i.e. a maximal subset of $targets(t)$ such that:

$$\forall s, s' \in targets(t).s \searrow^+ s' \Rightarrow s' \notin targets'(t)$$

Observation 2.22 (Scope Properties). *The following simple properties hold for individually computed scopes of targets of a single transition:*

1. *The implicit scope is always an or-state.*

2. *$\forall \sigma, t. \ |scope(\sigma, t)| \leq |targets(t)|$.*

3. *The generalized scope is an orthogonal set.*

Proof. The first property follows from the fact that only and-states are targets and from definition of implicit scope. The two other facts can be proved from static correctness conditions for transitions. □

The individual scope semantics is more similar to the semantics typically applied to flat (non-hierarchical) state machines, a well established specification language of the industry. Also it meets a typical need of communicating with other components in a cheaper way, than is possible with message passing. The UML like collective-scopes semantics can be implemented using individual scopes, by adding an additional target, forcing a single global scope (in the example of Fig. 2.3 this amounts to adding state A to the transition in question). The individual scopes semantics can be mimicked in UML only by means of rather heavy weight multiplication of transitions.

The single-transition firing relation relates a transition t, the current state configuration σ_0, store ϱ_0, history marking η_0 and a signal queue q_0 with a new state, store, history marking and the new signal queue. This means that while the transition is fired, a new state configuration may arise, some variables may be modified, the history marking may be updated, and some signals may be issued. The relation is defined by combined application of formerly specified exit, exec and enter relations.

$$
\xrightarrow[\text{fire}]{} \subseteq Trans \times \Sigma_{root} \times Store \times History \times Queue \times
$$
$$
\times [Ainst] \times \Sigma_{root} \times Store \times History \times Queue \qquad (2.60)
$$

$$
scope(t, \sigma_0) = \{s_1, ..., s_k\} \ \wedge \ \forall i,j \in \{1..k\}.\ i < j \Rightarrow \neg(s_j \blacktriangleleft s_i)
$$
$$
\forall i \in \{1..k\}.\ \langle s_i, \sigma_0 \cap descend^*(s_i), \varrho_{i-1}, \eta_{i-1}, q_{i-1}\rangle \xrightarrow[\text{exit}]{os_{i-1}!} \langle \varrho_i, \eta_i, q_i\rangle
$$
$$
\langle\langle action(t), \varrho_k, q_k\rangle\rangle \xrightarrow[\text{exec}]{os_k!} \langle \varrho_{k+1}, q_{k+1}\rangle
$$
$$
\forall i \in \{1..k\}.\langle targets(t) \cap descend^*(s_i), s_i, \varrho_{k+i}, \eta_k, q_{k+i}\rangle \xrightarrow[\text{enter}]{os_{k+i}!} \langle \sigma_i, \varrho_{k+i+1}, q_{k+i+1}\rangle
$$
$$
\sigma_{k+1} = \sigma_0 \setminus \left(\bigcup_{j=1}^{k} descend^*(s_j) \right) \cup \left(\bigcup_{j=1}^{k} \sigma_j \right)
$$
$$
os = cons^*(\textstyle\prod(os_0, \ldots, os_{k-1}), cons^*(os_k, \textstyle\prod(os_{k+1}, \ldots, os_{2k})))
$$
$$
\overline{\langle t, \sigma_0, \varrho_0, \eta_0, q_0\rangle \xrightarrow[\text{fire}]{os!} \langle \sigma_{k+1}, \varrho_{2k+1}, \eta_k, q_{2k+1}\rangle}
$$
$$
(2.61)
$$

The following claim together with Prop. 2.17 justifies well-formedness of configuration σ_{k+1}:

Proposition 2.23. *Let σ_0 be a state configuration of **and**-state s and $s' \in children(s)$. If σ' is a state configuration of s' then*

$$
\sigma_1 = \sigma_0 \setminus descend^*(s') \cup \sigma'
$$

is also a configuration of s.

Proof. Divide σ in configurations for children of s using proposition 2.17. Then exchange one of them for σ'. The union is a state configuration (Prop. 2.19). □

Theorem 2.24. *Let σ_0 be a state configuration of **and**-state s and state $s' \in descend(s)$. If σ' is a state configuration of s' then*

$$
\sigma_1 = \sigma_0 \setminus descend^*(s') \cup \sigma'
$$

is itself a state configuration of s.

Proof. Generalize proposition 2.23 by induction on depth of s'. □

2.2.7   Scheduling Transitions

Definition 2.25. *A transition t is enabled by an event instance $e(v_1, ..., v_k) \in Einst$ in state configuration σ and store ϱ iff*

$$e = event(t) \wedge (\sigma \models guard(t)) \wedge (\varrho \models expr(t)),$$

Note that the assumption of purity of all expressions plays a crucial role in the definition of satisfaction. This guarantees that whatever order we take to iterate over the transitions in the actual implementation, the same transitions will be considered enabled. Moreover, this would permit an optimized implementation to duplicate or skip the evaluation of expressions, without affecting the variable store. Nevertheless, this is insufficient to guarantee determinism of the enabledness. C functions called in boolean expressions may (and normally should) refer to external properties of devices, which in turn are dynamic in time. Only the assumption of synchrony hypothesis [10], that all guards are computed infinitely fast, can achieve a deterministic computation of the *enabled* set.

Two transitions are in conflict if they can be simultaneously enabled and they have overlapping scopes.

Definition 2.26. *Two distinct transitions t_1, t_2 are in conflict in a given state configuration σ and store ϱ iff both are enabled and*

$$\exists s_1 \in scope(t_1), s_2 \in scope(t_2). \ s_2 \searrow^* s_1 \vee s_1 \searrow^* s_2$$

IAR visualSTATE disallows conflicting transitions, while UML [98] and Harel [42] propose various conflict resolution strategies. Harel assigns a higher priority to a transition with a source state higher in the hierarchy. UML does the opposite [98][103, sec.15.3.12, p.493]. We model all these choices by using the priorities order on transitions $\lhd$. We assign priorities to transitions by means of a partial ordering $\lhd \subseteq Trans \times Trans$ imposed on the *Trans* set.

Two transitions are in a *resolvable conflict* if they are in conflict and they are comparable in the priority ordering $\lhd$. We always pick up the one that has higher priority (greater in the $\lhd$ order) and disregard the other one. All remaining conflicts are unresolvable. Unresolvable conflicts lead to nondeterministic semantics. We write $enabled(e(v_1, \ldots, v_k), \sigma, \varrho)$ to mean a maximal set of enabled transitions not in an unresolvable conflict (in the sense that adding any other transition would cause an unresolvable conflict). Note that this set may not be uniquely determined if the model contains unresolvable conflicts.

A single microstep of execution constitutes of firing all transitions that are enabled by an event instance e in a given state σ_0 and store ϱ_0. It also takes care of performing a substitution of actual event instance parameter values v_j for parameter names x^i_j.

$$\xrightarrow[\text{micro}]{} \subseteq Einst \times \Sigma_{root} \times Store \times History \times Queue \times$$

$$\times [Ainst] \times \Sigma_{root} \times Store \times History \times Queue \qquad (2.62)$$

$$\frac{\begin{array}{c} enabled(e, \sigma_0, \varrho_0) = \{t_1, ..., t_k\} \ \wedge \ \forall i, j \in \{1..k\}. \ i < j \Rightarrow \neg(t_j \lhd t_i) \\ \forall i \in \{1..k\}. \langle t_i[v_1/x_1^i, ..., v_m/x_m^i], \sigma_{i-1}, \varrho_{i-1}, \eta_{i-1}, q_{i-1} \rangle \xrightarrow[\text{fire}]{os_{i-1}} \langle \sigma_i, \varrho_i, \eta_i, q_i \rangle \end{array}}{\langle\langle e, \sigma_0, \varrho_0, \eta_0, q_0 \rangle\rangle \xrightarrow[\text{micro}]{\prod(os_0, ..., os_{k-1})!} \langle\langle \sigma_k, \varrho_k, \eta_k, q_k \rangle\rangle}$$

$$(2.63)$$

where $\langle x_1^i, ..., x_m^i \rangle = params(t_i)$ and substitution on transitions is naturally understood as substitution in guard and action expressions.

A *macrostep* is a chain of microsteps initiated by a single external event. After performing the microstep for the external event the microsteps are reiterated over internally signaled events until the system reaches stability (the signal queue is empty). Not surprisingly rules 2.64–2.65 resemble while-loop execution rules for imperative languages: a macrostep is usually implemented as a while loop performing microsteps.

$$\xrightarrow[\text{macro}]{} \subseteq \Sigma_{root} \times Store \times History \times Queue \times [Ainst] \times \Sigma_{root} \times Store \times History$$

$$(2.64)$$

$$\frac{}{\langle \sigma_0, \varrho_0, \eta_0, \langle\rangle \rangle \xrightarrow[\text{macro}]{\perp!} \langle \sigma_0, \varrho_0, \eta_0 \rangle}$$

$$\frac{\langle e(v_1, ..., v_m), \sigma_0, \varrho_0, \eta_0, q_0 \rangle \xrightarrow[\text{micro}]{os_1!} \langle \sigma_1, \varrho_1, \eta_1, q_1 \rangle \quad \langle \sigma_1, \varrho_1, \eta_1, q_1 \rangle \xrightarrow[\text{macro}]{os_2!} \langle \sigma_2, \varrho_2, \eta_2 \rangle}{\langle \sigma_0, \varrho_0, \eta_0, \langle e(v_1, ..., v_m)\rangle \hat{\ } q_0 \rangle \xrightarrow[\text{macro}]{cons^*(os_1, os_2)!} \langle \sigma_2, \varrho_2, \eta_2 \rangle}$$

$$(2.65)$$

Finally the global transition relation is:

$$\frac{\langle \sigma_0, \varrho_0, \eta_0, \langle e(v_1, ..., v_m)\rangle \ \rangle \xrightarrow[\text{macro}]{os!} \langle \sigma_1, \varrho_1, \eta_1 \rangle}{\langle \sigma_0, \varrho_0, \eta_0 \rangle \xrightarrow{e(v_1, ..., v_m)? \quad os!} \langle \sigma_1, \varrho_1, \eta_1 \rangle}$$

$$(2.66)$$

This transition relation explicitly shows the input and output of the system, emphasizing the synchronicity: the next environment input cannot be processed, before the reaction to the previous one is terminated. We will model this kind of systems using IO-alternating transition systems in chapter 5.

It can be observed that according to the above semantics, statecharts are *input-enabled*, i.e. for any reachable global state $(\sigma_0, \varrho_0, \eta_0)$ and for every

instance $e(v_1, \ldots, v_m)$ of environment event $e \in Event \setminus Signal$ there exist a reaction *os* and a subsequent global state $(\sigma_1, \varrho_1, \eta_1)$ such that:

$$\langle \sigma_0, \varrho_0, \eta_0 \rangle \xrightarrow{e(v_1,\ldots,v_m)? \quad os!} \langle \sigma_1, \varrho_1, \eta_1 \rangle \tag{2.67}$$

This follows from the fact that even if the set of the enabled transitions is empty, the microstep is still performed, producing $\bot$ (see 2.63).

2.2.8 System Initialization

The initial state configuration is not given in the concrete (visual) syntax, but must be derived from the initial marking (definition 2.5). The system must be *initialized*, that is, entry actions of and-states must be executed while building the initial state configuration. This happens by execution of the enter relation for the *root* scope with an empty explicit target set:

$$\langle \emptyset, root, \varrho_0, \eta_0, \langle \rangle \rangle \xrightarrow[\text{enter}]{os!} \langle \sigma_1, \varrho_1, q_1 \rangle \tag{2.68}$$

where ϱ_0 is an initial variable environment (part of the system) and $\eta_0 = ini\big|_{his}$ is the initial history marking. Before the execution may proceed with first external event instance, events signaled during the initialization step should be processed:

$$\langle \langle \sigma_1, \varrho_1, \eta_0, q_1 \rangle \rangle \xrightarrow[\text{macro}]{os'!} \langle \langle \sigma, \varrho, \eta \rangle \rangle \tag{2.69}$$

The state configuration σ, variable store ϱ and history marking *his* constitute the initial global state for processing of the first external event.

$$\xrightarrow[\text{init}]{} \subseteq [Ainst] \times \Sigma_{root} \times Store \times History \tag{2.70}$$

$$\frac{\langle \emptyset, root, \varrho_0, \eta_0, \langle \rangle \rangle \xrightarrow[\text{enter}]{os_0!} \langle \sigma_1, \varrho_1, q_1 \rangle \quad \langle \langle \sigma_1, \varrho_1, \eta_0, q_1 \rangle \rangle \xrightarrow[\text{macro}]{os_1!} \langle \langle \sigma, \varrho, \eta \rangle \rangle}{\xrightarrow[\text{init}]{cons^*(os_0, os_1)!} \langle \sigma, \varrho, \eta \rangle} \tag{2.71}$$

2.3 Related Work

Harel's enjoyable and easy-going introduction of statecharts [42] undoubtedly contributed to the significant success of the language. Notwithstanding, a side-effect of this informality was the emergence of a whole family of statechart dialects, with various features and purposes. The list includes original STATEMATE statecharts [45], Selic's real-time and object-oriented

ROOMcharts [117], Leveson's RMSL [77], Maraninchi's perfectly synchronous microstep-less Argos [86, 89] and OMG's UML statechart diagrams [102, chapt.15]—just to mention the best known. Already in 1994 von der Beeck reported on 20 available language variants. Since then the situation has clearly developed, but Beck's survey [128] remains a good introductory reading on design choices in statechart semantics even 10 years later. Another good reference on design choices is Huizing's and Roever's [55].

Unsatisfactory mathematical properties of the first formal definition [47] initiated a still ongoing, and apparently never ending, scholarly dispute on providing the right semantics for the language. A list of references on this, otherwise narrow and specific, topic is truly impressive: [56, 108, 87, 88, 93, 92, 23, 79, 78, 83, 124, 33, 129, 67, 74, 75, 76, 61, 136, 85, 25, 111]. Since our goal in this chapter was more pragmatic than innovative, we cease to discuss each of these semantics in detail. Instead of analyzing mathematical beauty or lack of thereof in the existing definitions, we propose a pragmatic survey, describing practical differences between variants instead of differences in the ways they were defined. We compare visualSTATE statecharts against two major variants: Harel's original statecharts and UML state diagrams.

2.3.1   Harel's Statecharts

We shall treat Harel's initial introduction [42] together with the subsequent formal definitions: Harel et al. [47] and the coinciding denotational definition of Pnueli and Shalev [108]. Harel's language was significantly more feature-rich than our semantics. Nevertheless most of his additional constructs (junction transitions, initial transitions, history transitions, do reactions, compound events, generated events etc.) can be modeled as syntactic sugar in our definition.

The output structure of original statecharts was based on sets, roughly corresponding to our semantics parameterized with values of the first row of Table 2.1. Similarly to us, they used microsteps internally, but these microsteps were not clearly visible from outside. This is also the property of our semantics instantiated for set output (the ordering of microsteps becomes visible in the sequence version though). The semantics of firing transitions is based on collective scopes as in UML, in contrast to visualSTATE's individual scopes (see section 2.2.6).

Harel allowed presence of more than one event at a time. Events were placed in a "pool" (a set), from which they were nondeterministically selected for processing. The pool played the role of signal queue in the visualSTATE dialect. Guards were enriched with conditions on contents of the pool, but events could not carry parameters, as they can in newer statechart variants.

The original paper suggested a conflict resolution strategy, by prioritizing transitions with respect to the depth of the nearest enclosing state. This

corresponds to selecting the transition priority order $\lhd$ to be induced by the hierarchy relation on collective scopes of transitions. Subsequent formalizations however, left out the problem of conflict resolution, in favor of allowing nondeterministic models, which can be modeled in our framework by choosing the priority order $\lhd$ to be empty. Harel's definition of conflict is also somewhat more fine-grained than ours, encompassing not only conflicting scopes, but also possible concurrent assignments to the same variables. This means that when two enabled transitions assign to the same variable, only one of them will fire. We find this solution very unreliable, instead advocating model checking technology [82] for statically detecting such situations in models statically and them as errors.

Similarly to our semantics neither of the two formal definitions was compositional, and both struggled with the problem of so called *schizophrenic models*, which can report on presence and absence of a specific output in the same step (the set of actions and events are identical in this model, which without special protection allows writing "schizophrenic" transitions like $s \xrightarrow{[\neg a]/a} s'$).

2.3.2 UML Statechart Diagrams

UML's statechart diagrams incorporated many features from Selic's ROOM-charts [117], statecharts of STATEMATE [45] and RHAPSODY [44]. They remain the most popular variant of statecharts—the one which continues to attract industrial tool developers. Our models can be seen as written in a subset of UML statecharts language. UML deliberately lacks a standardized formal definition. The UML standard [103, 102] remains an informal description sketching the notation and main semantic requirements for compliant tools.

The main difference between Harel's statecharts and UML statechart diagrams is their object-orientedness. UML statecharts describe behaviors of communicating objects, providing ways for synchronous and asynchronous communication. They abstract from details such as the synchrony hypothesis, deferring them to the actual application domains and tool developers. Consequently the definition of UML devotes considerably more space to syntax and static semantics, than to details of behaviors.

Nevertheless several characteristic points can be emphasized: UML statecharts are sequence oriented, allow truly concurrent transitions (our transitions are atomic and cannot be interleaved with each other) and propose conflict resolution based on depth of transition's scope. These correspond roughly to parameterizing our semantics with values of the last row in Table 2.1, an empty state priority order $\blacktriangleleft$ and a transition priority ordering $\lhd$ induced by the reverse of substate relation on transition scopes. In contrast to Harel, UML prioritizes transitions which are nested deeper in hierarchy, claiming that this reflects object-oriented principle better. This argument is

somewhat unfortunate, since neither of the choices guarantees preserving of any reasonable refinement relation, if higher priority transitions are added. More interestingly the UML specification proposes a runtime implementation of conflict resolution based on a greedy search. In section 3.4.2 we shall see how this costly idea can be eliminated in favor of a static algorithm.

Despite allowing asynchronous communication (message passing) UML statecharts react to events synchronously: only one event is processed at a time, very much in the spirit of our definition. UML does not formalize the notion of an event queue, using an event pool instead. Various selection strategies can be proposed in actual tools [103, p.491]. Priority-based schemes cannot be easily expressed in our semantics without any extensions.[2]

Similarly to Harel's statecharts, UML statecharts are richer than our model, but many of missing constructs can be easily simulated via syntactic expansions. Things that cannot be achieved are parameterized internal events (explicitly disallowed in our model) and multiple objects communicating. There are also some terminology differences: LCA (the lowest common ancestor) is used instead of *NCA*, a macrostep is usually referred to as a *run-to-completion* step [103, p.491].

There is a vast number of attempts to formalize UML in mathematical terms, especially UML statechart diagrams [79, 78, 124, 33, 118, 129, 67, 74, 75, 76, 61, 68]. These definitions do not coincide, as they make conflicting choices for unspecified aspects of the language.

2.4   Summary

We have presented a formal definition of IAR visualSTATE statecharts. Our goal was pragmatic, namely to give a solid and precise definition, rather than innovative (to investigate new ways of formalization). We have given a global non-compositional big-step semantics for statecharts, which will be used later on as correctness specification for model transformations and optimizations.

Our semantics was parameterized with the output-structure of the models and priority ordering on states and transitions. This way we have been able to emphasize these differences between language dialect, which will become important in later developments. We have observed that the semantics is input-enabled, reactive, and synchronous, and that various output structures describe the level to which subtleties in the semantics are observable to the outer environment. This parameterization has given us a crude way to control observability of reaction changes. A code generator can freely choose the ordering of firing transitions as long as the model of output structure

---

[2]Fixed number of priority levels can be implemented by modeling several queues using a single queue, which is possible theoretically, but hardly feasible in realistic applications.

does not perceive the obtained output as different (a lot of reordering with allowed in the set-based outputs). In chapter 5 we will discuss an advanced framework for fine grained control over permissible changes in code generators, that would not only allow reordering but also dropping and renaming some outputs.

A code generation algorithm is correct if for every input statechart the behavior of the produced program is a correct refinement of the formal behavior of the model. For input-enabled synchronous systems many behavioral refinement relations of the so called van Glabbeek's spectrum [126, 127] coincide. In chapter 5 we shall discuss new notions of refinement interesting from the perspective of code generation.

3

Code Generation Overview

We shall now introduce the problem of code synthesis from statecharts and outline the most typical solutions. We explain the general architecture of our code generator and detail some preparatory steps that we undertake before tackling the core problem. After reading this chapter the reader will know all the details required by our code generation algorithms, themselves presented in Chapter 4.

We begin with a brief statement of the problem in Section 3.1. Then, in Section 3.2, three general approaches to code generation are described; our work is of the third kind. Section 3.3 gives an architectural overview of our tool, SCOPE, and also sheds some light on the methods which we used to evaluate its efficiency. Section 3.4 focuses on model transformations applied by SCOPE's front-end. Section 3.5 discusses related work on transformations of statecharts and mentions available surveys of code generation methods.

If you are predominantly interested in how SCOPE is built, you should read sections 3.1, 3.3–3.4 and continue with chapter 4. These are also the very sections of the present chapter that contain direct contributions. If you are reading this text as part of a more general study of code generation methods, you should not miss sections 3.2 and 3.5 either, for they report on the work done by others.

3.1 Requirements

In the center of our interest is synthesis of programs from synchronous statecharts. Such synthesis furnishes a platform independent program capable of input and output: it can receive events (though the glue code generating events needs to be provided) and can call the action functions to realize the semantics of the model. We shall require that the generated program conforms to the original model, implementing the macrostep semantics as given in chapter 2. In chapter 5 we will be more specific on what the nature of this conformance could be. Ultimately, we will demonstrate that

controlled relaxation of the conformance requirement leads to interesting practical applications in development of families of control programs.

Following the *de facto* standard in development of embedded systems, we shall use C [59] as our target language. This allows easy interfacing from our code to most legacy drivers for sensors and actuators. The generated program takes the form of a *sequential* kernel, which for submitted events advances the internal state and calls action functions. The programmer, the ultimate user of our tool, should provide the implementation of actions and guards in C fulfilling the requirements of our semantics: guards should be pure and actions should be relatively pure (their side effects cannot be interdependent). This guarantees that the system remains insensitive to variations of interpretation order for transitions and states. The running time of all the user provided code must be negligible. Otherwise the synchrony hypothesis cannot be maintained (see section 2.2.7).

Despite the use of a standardized target language, we still need to indicate our execution platform. Our programs shall run on popular sequential micro-controllers, commonly embedded in small electronic devices, for example Atmel's AVR/ATmega series (`http://www.atmel.com/products/avr/`), or Hitachi's H8/300 family (`http://www.renesas.com`). We want to challenge the domination of low-level programing languages (assembly languages and C) on these platforms. We are not directly interested in influencing the development processes on bigger platforms such as ARM or x86, where high-level languages are already deployed with great success. Occasionally, we will report results of experiments on the Intel x86 platform, because this is the easiest one to relate to for other researchers. Bear in mind though that we are not interested in targeting Intel x86 as such.

Typically, the amount of memory available in an embedded application of interest will vary between 8 and 64 kilobytes. Most of the projects that we have seen enjoyed 16 kilobytes of total memory. In many cases most of the memory would not be writable, leaving only one or two kilobytes for dynamic data. Obviously most of the memory should not be used by the control kernel, but should be left available for the developer of the application, for his own data and code. All in all it is essential that the code we generate uses as little memory as possible, and is especially conservative about writable memory. This rules out the use of dynamic memory management and implies a restricted usage of the hardware stack, avoiding parameters to function calls and calls themselves whenever possible.

3.2   State of The Art

Let us briefly examine three major classes of approaches to code generation from statecharts and discuss their suitability for embedded applications with constrained resources.
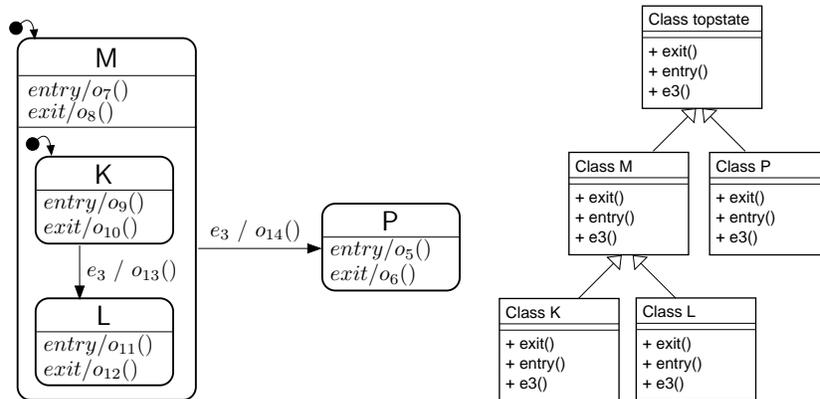
Figure 3.1: A state pattern example: (left) a sample statechart, and (right) a class hierarchy created by using the state pattern to implement it.

3.2.1   State Pattern Group

The *state pattern* has been popularized by the well known text of Erich Gamma and others on design patterns [39, pp. 305–313], and attributed to Johnson and Zweig [62] therein. Its application to statecharts has been sketched in the main UML manual of Booch et al. [17]. It encapsulates behavior specific to each state in a separate class. The hierarchy tree of the model is encoded in the object-oriented type system: a class representing a given state is an extension of the class representing its parent state.

Figure 3.1 illustrates a statechart and the respective class hierarchy. Note that the class hierarchy resembles the shape of the statechart's decomposition tree induced by the substate relation (an example of such a tree, albeit for a different statechart, was shown in figure 2.1). Figure 3.2 gives a complete implementation of this statechart in C++. Current state is always kept as an object pointed to by the static field `topstate::current`. The methods implementing transitions are named after the firing event: only `e3` in this case. Each of such methods allocates a new object for the target state and points `topstate::current` to it. Deallocation is left to the caller as deallocating an object inside its very method call is unsafe in C++ (in Java-like languages this would be left to a garbage collector). If a given state does not handle the `e3` event then its `e3()` method should return zero. The caller code is supposed to recognize this case and not attempt to deallocate the object in such case.

Figure 3.3 shows a simple driver that, after having initialized the program, continues to send the e_3 event to the statechart indefinitely. The initialization comprises creation of the object for the initial state K and calling the required entry methods. In realistic applications this loop needs to interface sensor drivers, either by polling the sensors or by checking some

```
class topstate {
public: virtual void entry()=0;
        virtual void exit()=0;
        virtual topstate * e3() { return 0L; }
protected: static topstate * current;
friend int main(void);
};

class P :public topstate {
public: virtual void entry() { o5(); }
        virtual void exit()  { o6(); }
};

class M :public topstate {
public: virtual void entry() { o7(); }
        virtual void exit()  { o8(); }
        virtual topstate * e3() {
                    this->exit();
                    this->M::exit();
                    o14();
                    (current = new P())->entry();
                    return this;
        }
};

class L :public M {
public: virtual void entry() { o11(); }
        virtual void exit()  { o12(); }
};

class K :public M {
public: virtual void entry() { o9();  }
        virtual void exit()  { o10(); }
           topstate * e3() {
               exit();
               o13();
               (current = new L())->entry();
               return this;
           }
};
topstate * topstate::current = 0L;
```

Figure 3.2: An implementation of the state pattern for statechart of Fig. 3.1

```
int main (void) {
  K * temp = new K();
  temp->M::entry();
  temp->K::entry();
  topstate::current = temp;
  while (1) {
    topstate * temp = topstate::current->e3();
    if (temp) delete temp;
  }
}
```

Figure 3.3: A C++ driver for statechart implemented in Fig. 3.2

buffers for events generated by sensor drivers in concurrent threads.

Note that whenever a new state is entered, a new object is created. This object, by means of virtual calls, performs a conflict resolution in UML style. With UML style conflict resolution, the first e_3 instance is handled by the transition between K and L, while the second occurrence of e_3 enables the transition sourced in M. The object-oriented type system automatically chooses the right transitions, by choosing the respective overridden implementation of the e3 method.

The major weaknesses of the state-pattern are its difficulties with supporting history states and concurrency. It has been extended by Ali and Tanaka in [1] to handle both features, unfortunately at the cost of losing its most powerful advantage: the neat encoding of the state structure in the object-oriented type system. In the extended version all classes representing or-states are aggregated into the classes implementing and-states. This breaks the "magic" transition selection by virtual method calls, increases the size of the program and decreases execution speed.

In general heavy use of object-orientation (or use of object-orientation at all) is not suitable for the constrained applications that we consider. Massive growth of memory consumption is the price paid for the nice abstractions. Since we intend to generate the kernel code automatically, without supporting direct user interventions in this code, we would hardly benefit from the abstractions, only pay their high price. The executable produced by compilation of the C++ program of Fig. 3.2 and the `main` function presented on Fig. 3.3 produces an executable of 4348 bytes (x86 platform, GCC ver. 3.3.4, optimizing for size, *dynamically* linked and stripped). Dynamic linkage means that the size of memory manager is *not* included in this number, but it is still needed when the application is deployed. In chapter 4 we will show that a less direct interpreter-based method is able to achieve this executable size for much bigger and concurrent models, even with *static* linking. Moreover the big chunk of this executable will be used up by the runtime

interpreter, which is independent of the model size, meaning that the other method will scale much better for big models.[1]

An implementation of the state-pattern in a non object-oriented language can be realized by use of nested switch statements [17, p.338]. In this case the programmer explicitly maintains information about the current state, and uses switch statements instead of virtual functions to resolve state-dependent behavior. Fig 3.4 demonstrates such implementation for the example of Fig. 3.1. These two methods are nearly identical when analyzed from the perspective of the native code executed in the end (switch statements are usually compiled to dispatch tables—general kind which also encompasses virtual tables used for dispatching virtual methods in object-oriented languages). The nested-switch approach uses less memory than the original object-oriented scheme (3448 bytes in this case).

Pinter and Majzik [106] attribute the nested-switch-statement technique to the Rhapsody tool distributed by I-Logix (albeit much improved). Somewhat contradictory Zündorf [138] claims that Rhapsody uses the object-oriented state pattern, while one of its major competitors, Rational Rose (presently an IBM product), uses switch statements.

3.2.2 Samek's Quantum Programing Framework

In the only published book [114] on translating statecharts to code, Miro Samek, proposes an improvement of state pattern, using the name of *quantum programing*.[2] Instead of using objects to represent states, he uses pointers to event handlers. The gain is that dispatching an event to a state is extremely cheap: one pointer indirection and function call, which is compiled to a single-instruction indirect jump on many architectures. The price is that exit and entering need to be handled using special events by the same handlers as all the usual environment events. Samek does not represent hierarchy explicitly neither in the data structure, nor in the type system. The event handlers contain nested calls to substate event handlers.

The quantum framework does not explicitly support concurrency, so the book presents a way to achieve simple model of concurrency without modifying the framework. Each statechart is primarily sequential, but it may incorporate objects. If each event handler takes care to forward events to these components, then a concurrency-like effect is achieved. This happens at the cost of code duplication: the dispatch to orthogonal components is present in each event handler of the superstate. This is both inconvenient

---

[1] The state pattern can be implemented without dynamic memory management, by pre-allocating all state objects needed statically and just redirecting the current state pointer to the existing objects. This might be cheaper for small models, but still relatively expensive due to the requirement of keeping all the objects constantly in memory. Typically the number of active states is much small than the number of all states in the model.

[2] The name is confusing because it has no relation to the area of *Quantum Computing*

```
void state_exit(state s) {
  switch (s) {
  case M: o8();  break;
  case K: o10(); break;
  case L: o12(); break;
  case P: o6();
  }
}

void state_entry(state s) {
  switch (s) {
  case M: o7();  break;
  case K: o9();  break;
  case L: o11(); break;
  case P: o5();
  }
}

void e3() {
  switch (current) {
  case L: state_exit(L);
          state_exit(M);
          o14();
          state_entry(P);
          current = P;
          break;
  case K: state_exit(K);
          o13();
          state_entry(L);
          current = L;
  }
}

int main (void) {
  current = K;
  state_entry(M);
  state_entry(K);
  while (1)
    e3();
}
```

Figure 3.4: An implementation of the nested-switch variant of the state pattern (in C). Includes the driver in the main function.

for manual maintenance and expensive when code generation is used.

Nevertheless Samek claims that his method is the fastest in practice. At the same time Pinter and Majzik report that it is known to produce big executables [106]. They show a simple example of a statechart without concurrency, taken from the [114] for which the size of the executable implemented in Samek's quantum framework exceeds the size of the code generated with their code generator (EHA2C) by three times. Small experiments performed in our project lead to similar observations. See appendix A for an implementation of the statechart of Fig. 3.1. This statechart, though extremely simple, still takes more than six kilobytes after compilation.

3.2.3   Interpretation Approach

An alternative approach to code generation relies on building a runtime representation for the model and then providing a static interpreter for it. The solutions from this group tend to be slower than generated native code, but they enjoy a number of advantages. Most prominently it seems that they are capable of yielding smaller code, with much lower writable memory consumption. This is considered a more critical property than speed, as in most statechart applications (user interfaces in embedded systems, protocols, etc) speed does not seem to be an issue[3]. Memory consumption is much more of a problem. The interpretation approach is employed by, among others, the IAR visualSTATE code generator, EHA2C [105], Fujaba [138] and the outcome of this thesis—the SCOPE code generator [137].

The interpretation approach is often used in generation of state machines in other areas of computer science, most notably various implementations of parser and lexer generators. Generation of native code is used where speed is a real issue—for instance in explicit state model checkers, like SPIN [53] (and in certain sense also in UPPAAL [6]). Explicit state model checkers perform an execution of finite state models as automata. For efficiency reasons the automata are first generated as C code then compiled and executed natively.

Pintér and Majzik [106] report results of experiments evaluating the fault tolerance of the direct approach (exemplified by the quantum framework) against the fault tolerance of the interpretation approach (represented by their tool EHA2C). They introduce faults by mutating single bits in the executable. They find that the interpreter-based method detects more faults, when only user-level fault detection mechanisms are available. The direct method more easily fails without detection, but then the memory protection mechanism takes over and efficiently detects more errors than the assertions of the interpreted code. They argue that this makes the interpreted method more suitable for small systems where memory protection is not available. This is exactly the class of systems we are interested in.

---

[3]As suggested by industrial partners.

3.2.4   BDD-based Encoding

Another alternative, different both from the simple interpretation and from the generation of native code, is to encode the statechart semantics in terms of more primitive constructs such as boolean equations or propositional formulæ. The former are used in compilation of synchronous languages [88]. Jacobsen [60] builds a propositional logics representation for statecharts. He encodes the reaction relation using Bryant's reduced ordered binary decision diagrams, also known as ROBDDs [18]. His encoding of the model closely resembles encodings used in model-checking of statecharts [82, 50]. He only handles flat state/event systems, which are similar to statecharts without hierarchy. We shall discuss flat statecharts in depth in chapter 4.

In short, Jacobsen splits the reaction of the system into two relations: one representing the state change and the other representing the outputs being produced. Then each transition is encoded as a conjunction of the representation of its firing condition and the representation of its action. The whole system is encoded by a combination of disjunctions and conjunctions of encodings of all its rules. With this encoding, finding the next state configuration and outputs is reduced to answering satisfiability questions: one needs to enumerate all satisfying paths in the BDD.

Splitting the reaction into two separate steps (advance the whole system and then return the outputs) differs from the standard UML semantics (run-to-completion step). However the difference is only visible internally. From an external point of view the system behaves correctly. Such a construction is also very close to the way in which hardware engineers perceive reactive systems. Note that this split of output and next-state relations is not necessarily expensive in size, since both relations will share the same variable space and the same subexpressions, thus efficiently using sharing properties of BDDs. There is a certain speed penalty in computing the relations twice, as they check the same or similar conditions. It seems that the solution can be easily adopted to the UML style (calling actions while transitions are fired), eliminating the additional speed overhead.

Jacobsen's thesis reports numerous variations of encodings and compares the generated code quantitatively with the one produced by visualSTATE. Unfortunately the results are not as good as expected, only rarely competing with the simple interpretive method used in visualSTATE. The BDD engine, which needs to be present at runtime, is typically larger than the interpreters used in more direct methods. To make things worse the typical BDD implementation performs logical operations on BDDs at runtime, on the embedded platform. This requires a considerable amount of writable memory, with very weak guarantees on the actual amount used at runtime.

All in all, the encoding approach, though scientifically appealing, still remains behind more direct solutions presented before. Perhaps it still awaits an investigator who can leverage its qualities. I am not aware of any similar

work applied to truly hierarchical systems. There are some doubts how to efficiently transform these results to fully-fledged statecharts, with guards in the host language, variables, and message passing.

3.3 Overview of **SCOPE**

Over the years of experiments our implementation, SCOPE, has grown from a simple code generator for visualSTATE statecharts to a development toolkit supporting various aspects of code generation. Nowadays it includes several language front-ends, several specific back-ends, several code-generation algorithms, format converters, visualizers, and a simple model checker. It is implemented in Standard ML [95], using the BDD package BuDDy [37, 80]. In this section we present its basic design decisions.

3.3.1 Input languages

The primary input language of SCOPE is the language of statecharts as described in chapter 2, with minor extensions. Being a visual language it requires also a textual representation. SCOPE uses the proprietary input format of the visualSTATE tool, as shipped in versions 4.x and 5.x. This is a textual format which is relatively easy to read for humans and which, unlike XMI [101], can be composed manually.

The ultimate standard format for textual representation of statecharts is XMI [101]—XML Metadata Interchange format—standardized by OMG. To the grief of tool builders, XMI enjoys several versions, compatible, or rather incompatible, with a handful of UML [98] and MOF [100] versions. As such XMI is hardly standard and hardly facilitates model exchange across tools at this time, as various tools use various combinations of standards.

Another weakness of XMI, namely imprecision of the semantics, is inherited from UML. UML designers decided to give a very general informal semantics of the language, allowing for its specific instantiations by users and tool vendors. Models transported with XMI do not have a well-defined semantics, and the hypothetical results of running SCOPE on the XMI input would not be directly comparable to results produced by other tools.

Nevertheless, XMI has been available for several years now and commercial tools start to incorporate it, bringing hope that it will become mature soon. It is expected that with the advent of UML 2.0 the XMI representation for statecharts will be greatly simplified. Unfortunately the new standard has not been published at the time of writing these words. I do hope that XMI support will be incorporated into upcoming versions of SCOPE.
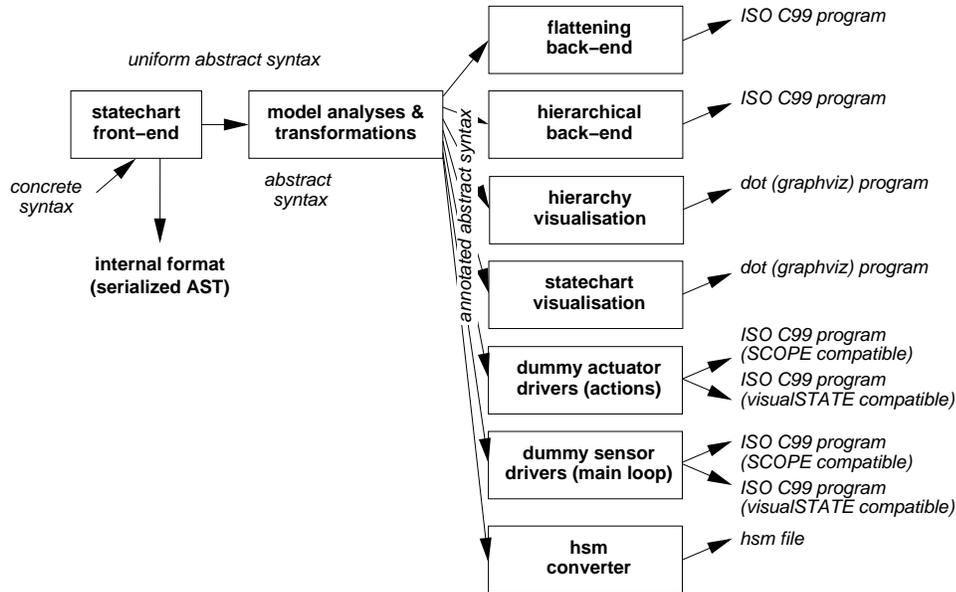
Figure 3.5: A simplified view of the architecture of SCOPE code generator

3.3.2   Output Languages

The main output language of SCOPE is ISO C [59]. A range of back-ends target C, most notably: the flattening and the hierarchical code generators (see chapter 4), the generators of dummy action and guard functions and the generators of dummy sensor drivers. Both generators of dummies are instrumental in testing and evaluating the main generators mentioned above.

Other back-ends include a generator of graphviz [40] scripts for visualization of models and decomposition trees, a converter to an internal binary format and a converter to the locally developed hsm format [104, 19] (a simple textual format for statecharts, easy to parse, useful for student projects).

3.3.3   Architecture

Figure 3.5 presents the structure of SCOPE's implementation, indicating three layers: a layer of front-ends, a layer of transformations, and a layer of back-ends. The front-end consists of a number of parsers; most notably of those that parse files in IAR visualSTATE file formats.

After the initial stage, the tool applies model transformations: elimination of dynamic scopes and annotation of transition targets with scopes. We shall explain this process in section 3.4.1. Most of the other transformations are very simple and not worth mentioning here. However, this part of the tool has been undergoing the most heavy development recently, to accommodate the results of chapter 6.

The back-ends of the SCOPE code generator make use of scope annotations on transitions to produce exiting and entering code, which operates without runtime scope computations. Additional back-ends provide visualization aid, by means of the *graphviz* package. These back-ends support testing and reverse engineering of models and are especially useful if a license to the visualSTATE development environment is not available.

Finally a range of back-ends producing dummy drivers is used for testing and evaluation of the generated code. Some of these back-ends produce dummy substitutes for all C functions called in the model (actuator drivers). These back-ends come in two kinds: one that generates completely empty functions with no dependencies on libraries; and one that generates functions sending observable outputs to standard output, which are then used for black-box testing. Naturally the empty implementations of actions are used whenever we compare the size of binaries produced of generated code, so that the size of libraries does not distort the outcome of comparisons.

Similarly, SCOPE provides back-ends that generate sensor code used in testing. There are three variants of these back-ends: a minimal one (for code size evaluation), a random one (for uncontrolled tests) and one translating events from standard input to model inputs (for controlled black-box tests). The minimal back-end is independent of any libraries. It calls the macrostep function in a non-terminating loop, assuming a fixed environment event as input. As with actuators, these back-ends come in two flavors: one compatible with SCOPE and one compatible with visualSTATE.

Given such an abundance of back-ends, it is clear that SCOPE is highly retargetable. In fact more back-ends have been planned, shortly before these lines were written. In future SCOPE will support a custom format used in development of thermostat controllers by Danfoss (see section 4.7 and [71]).

3.4   Model Transformations

3.4.1   Elimination of Dynamic Scopes

As we have mentioned in section 2.2.6, visualSTATE implements individual scope semantics exhibiting the problem of dynamic scopes. For some transitions, namely *dynamically scoped transitions*, it is impossible to determine the scope of the transition statically at compile time (see Fig. 2.3). We have modeled this theoretically by computing the value of *iscope* function *at runtime* (see definitions 2.20-2.21 and rule 2.61). This is not acceptable in implementations of resource-aware code generators, as it complicates the logics of the runtime interpreter significantly. It uses more space and slows down the interpretation of all transitions, also statically scoped ones.

Various optimizations can be applied to improve efficiency of scope computation. We think, however, that the most efficient solution is to transform the model to guarantee that all transitions have statically resolvable scopes

and then precompute scopes at compile time. In this case it is possible to remove all scope resolution support from the interpreter, making it simpler, more orthogonal and efficient for all of the transitions without loss of performance or increase of size in the frequent statically scoped case.

Let us give an intuition about the main idea of dynamic scope elimination. Recall the statechart of Fig. 2.2. As we have explained before, the scope of transition $D \xrightarrow{e_1 \ [D]/f_{15}()} \{E, I\}$ consists of two or-states: state B and a second state chosen from C or G', depending on the current configuration. We can split this transition into several mutually exclusive rules for which static scope resolution is possible:

$$D \xrightarrow{e_1 \ [D \wedge F]/f_{15}()} \{E, I\}, \text{ scopes: } B \text{ and } C \text{ respectively} \qquad (3.1)$$

$$D \xrightarrow{e_1 \ [D \wedge G]/f_{15}()} \{E, I\}, \text{ scopes: } B \text{ and } G' \text{ respectively} \qquad (3.2)$$

We proposed two rules instead of one and ensured that scope can be resolved statically for each of them by extending guards with extra conditions. In the following we discuss how to automate this task.

The scope of a transition is independent of current configuration if its guard contains a *branch excluding expression* over each of its targets:

Definition 3.1. *Let s_1 and s_2 be two distinct and-states. An expression $s_1 \wedge \neg s_2$ is called a branch excluding expression, or simply a branch exclusion, iff the parent of s_2 is a substate of s_1 ($s_1 \searrow^2 s_2$).*

It is not strictly necessary for a transition to contain a branch exclusion in its guard in order to enjoy static scopes. It is however necessary that branch exclusion is implied from transition guards and model structure. The following theorem summarizes the above intuition: a state change enjoys having a static scope if its guard combined with the model structure implies a branch exclusion expression. The scope is also static when the target is guaranteed to be included in the source configuration (a self-loop transition).

Theorem 3.2 (Branch-Exclusion). *Let ϕ be a formula overapproximating all state configurations of a given statechart model, but not greater than the set of all statically legal configurations. Assume that and-state s is among the targets of a transition t. The scope of a state change to s is static iff one of the following conditions is satisfied:*

1. *There exist and-state ancestors s_1, s_2 of s (possibly $s_2 = s$), such that $s_1 \searrow^2 s_2$ and $\phi \wedge \mathit{guard}(t) \Rightarrow (s_1 \wedge \neg s_2)$ or*

2. *$\phi \wedge \mathit{guard}(t) \Rightarrow s$,*

If the first condition is satisfied, then $\mathit{parent}(s_2)$ is the corresponding static scope. If the second condition is satisfied, the static scope is the parent of s.

Proof. (sketch for case 1). Take any legal configuration σ such that transition t is enabled in σ. Since $\sigma \models \phi$, the assumption implies a branch exclusion $s_1 \wedge \neg s_2$. So s_1 is the closest active and-state to s in σ. Since s_1 is active and it is an and-state, it must have active descendants, including descendants of *parent*(s_2). By definition 2.20 *parent*(s_2) is the scope of state change to s in σ. The proof in reverse direction proceeds in a similar manner. $\square$

The exclusion theorem helps to detect if a given transition is statically scoped. It does not solve the general problem of what to do if the transition is dynamically scoped. The way to proceed is to multiply each transition, extending its guard with suitable branch exclusions. Static resolution becomes possible and trivial for each transition decorated in this way.

The computation begins with building a formula ϕ representing a safe overapproximation of the reachable state space of the statechart in question (for example the set of all syntactically legal state configurations). For a given transition t we restrict the overapproximation to the set of states which enable t, and existentially quantify away all variables that do not represent targets, or their ancestors. We obtain an equation whose solutions are possible activity assignments when t fires; this in turn allows extraction of all scopes that are possible at runtime (or a little bit more if ϕ was indeed an overapproximation of reachable state space). The solutions of the system of equations may be obtained by means of a SAT-solver, or a BDD engine. Each separate assignment represents a single concrete transition being a part of the more abstract, dynamically scoped transition t.

$$\phi(t) = \exists s_1, \ldots, s_n. \; \phi \wedge \mathit{guard}(t),$$
$$\text{where } \{s_1, \ldots, s_n\} = \mathit{State} \setminus (\bigcup_{s \in \mathit{targets}(t)} \mathit{ancest}^*(s)) \; . \quad (3.3)$$

A transition is trivially unreachable if $\phi(t)$ is not satisfiable. Lack of solutions proves that the guard condition is contradictory and the transition may be safely discarded, perhaps issuing a warning.

Due to the hierarchical structure of configurations, the satisfiable assignments of $\phi(t)$ exhibit a regular pattern: each path down the hierarchy starts with some variables assigned true and switches permanently to false at some point. If there exists exactly one satisfiable assignment, then by the exclusion theorem, t has static scopes and the scopes can be inferred from the solutions of $\phi(t)$. One needs to identify the branch exclusion (the switch-point from true to false on the ancestors path) and use the exclusion theorem. Note that there is no need to extend guard conditions in this case. The existing guard is sufficient to guarantee desired properties of scopes.

Transition t is potentially dynamically scoped if $\phi(t)$ has more than one satisfying assignment. There exists such an assignment for each potential

scope. Each assignment may contain several branch exclusions, but at most one over each target. The branch exclusions may not be contradictory as they come from the same solution of $\phi(t)$. For each satisfiable assignment we create a new transition t_j extending the guard with branch exclusions found in that assignment. Guard simplification may be used to ensure that no redundant checks are introduced. Minimality of guards for newly created transitions is not guaranteed by the algorithm itself.

The transition cloning performed by elimination of dynamic scopes is not expensive in realistic situations. In a typical model there are only very few (if any) transitions with dynamic scopes, and only these transitions will be multiplied. At the same time much of the interpreter logic can be removed from the runtime code. Experiments confirm this intuition.

Last but not the least, two or more targets may share a common scope in a single transition. In such case the scope should only be exited and entered once, obviously. The targets should be classified in subsets tagged with common scopes. If the scope is dynamic it may happen that the grouping will depend on the current configuration. This problem is also solved by the above algorithm. Once static scopes have been inferred, the targets may be grouped into proper categories at compile time.

The middle layer of SCOPE implements the algorithm presented above using a BDD engine [80, 37]. We have not experienced any BDD explosion problems while applying it, using the set of all statically legal configurations as the approximation of reachable state space ϕ. The implementation was using about 2.5s to compile a model of about 200 transitions, on a Pentium III, 1GHz machine running Linux, including the cost of other passes of SCOPE. The static scopes algorithm is called once for each transition in the model. We believe that a direct algorithm, significantly less complex than SAT-solving, can be proposed for this problem, but given the availability of the BDD engine in the tool anyway, it was convenient to solve it in this way. In the end it proved to be fast, too.

3.4.2 Conflict elimination

We believe that there are more dynamic properties of statecharts that may be analyzed and precomputed at compile time with only very little memory cost at runtime. Another suitable transformation is the *conflict resolution* (see section 2.2.7). Detecting conflicts at runtime and finding the maximum set of non-conflicting transitions, optimized with respect to priorities, is a memory intensive task. Both the number of conflicting transitions and the size of resolved set are unknown. The resolution requires relatively complicated algorithms and data structures at runtime. Instead we would like to refine the transitions again, so that conflicts are ruled out from the model, while the original semantics is preserved. This is possible because conflict resolution relies on a static concept of priority, which is known at

compile time.

   Recall that two transitions are in conflict, if they can both be enabled at the same time, and they have targets in ancestrally related scopes (recall the definition of scope on p. 29). For the purpose of *conflict elimination* we assume that the scopes are only static. This can be obtained by applying the algorithm of the previous section first.

   Assume that ϕ is a formula overapproximating the reachable state space as before. Transitions t_i and t_j may be in conflict if the event triggering them is the same, $\phi \wedge guard(t_i) \wedge guard(t_j)$ is satisfiable, and the two transitions have ancestrally related scopes, i.e.:

$$\exists e, \sigma, \varrho.\ \exists s_i \in scope(\sigma, t_i).\ \exists s_j \in scope(\sigma, t_j).$$
$$t_i \in enabled(e, \sigma, \varrho) \wedge t_j \in enabled(e, \sigma, \varrho) \wedge (s_j \searrow^* s_i \vee s_i \searrow^* s_j) \quad (3.4)$$

Assume, without loss of generality, that $t_i \lhd t_j$, so the priority of t_j is higher than t_i. To eliminate the potential conflict between the two transitions we must remove from the set of configurations enabling t_i those configurations that enable also t_j. This can be easily achieved by refining the guard of t_i to be $guard(t_i) \wedge \neg guard(t_j)$. At this stage it is sufficient to do this refinement syntactically. Guard minimization can simplify the resulting expressions in the back-end later on.

   Let n be the number of transitions, $n = |Trans|$, and the number of literals in each single guard be bounded by some constant $O(1)$. This is a perfectly reasonable assumption for realistic models, where guards never refer to all model components, even if the model is very big. Our static conflict resolution algorithm may extend the guard of each transition with $(n-1)O(1)$ new literals, during refinement. As a consequence the entire model will grow not more than $O(n)$. In practice the rate of growth depends on the model itself and the accuracy of ϕ.

   SCOPE does not currently implement static conflict resolution, but it would be a straightforward extension to make it so. This choice directly follows the design decision of its elder commercial relative. IAR visualSTATE encourages developers not to rely on runtime conflict resolution, because it only slightly contributes to succinctness of models, and the beauty of its semantics is questionable. Most importantly, it may be prohibitively expensive for constrained embedded systems. Instead of supporting conflict resolution, visualSTATE chooses to detect conflicts during model checking. All reachable conflicts are reported as errors.

3.5   Related Work

We have already mentioned von der Beeck's extensive survey of statechart variants [128]. Unfortunately there is no such comprehensive account of applied work on statecharts: neither for model-checking nor code generation.

Existing surveys of code generation methods are rather superficial, but I recommend the respective parts of [114, 138, 106]. The real comprehensive survey with experimental comparisons still awaits a brave author. To do it well, one would have to implement all the known algorithms.

As an experiment we have implemented an interpreter for statecharts in Standard ML that interprets the abstract syntax trees of SCOPE, following the semantics rules very closely. Needless to say the algorithms employed in the interpreter were complex and unacceptable for compact C programs. This led us to static model transformation algorithms presented above.

The algorithm for elimination of dynamic scopes of section 3.4.1 was originally published by us in [135]. Algorithms for static conflict resolution, similar to the one of section 3.4.2, were reported independently by Diethers and colleagues [28], and Holcombe and Bogdanov [14, 15]. Both attempts applied this technique in the area of model validation (model checking and testing). The present formulation, sadly reinvented again, is the first one applied directly to code generation, presented as a standalone model transformation rather than a part of sophisticated process of encoding into some verification formalism.

We have also experimented with Java as a target platform. Pihl, Berger and Gram, all students of IT University in Copenhagen, implemented a code generator from a subset of statecharts targeting Java during a short term project [104]. I personally implemented some simple models in Java manually using the state pattern approach. Finally Steensgaard-Madsen and myself created a tiny code generator [19] for a subset of statecharts in Dulce [120, 121]—a framework for writing lightweight interpreters. In all cases the size of the Java class file produced was comparable to the size of the statically linked binary produced via C-based code generators. Given the fact that Java implementation requires a considerable burden of the virtual machine on all but very few embedded platforms, we have abandoned this direction for now.

Due to our focus on discrete systems with only soft time requirements, we have consistently ignored the technical difficulty of maintaining the synchrony hypothesis. These problems are normally considered orthogonal to the problems of efficient runtime representation and execution. Nevertheless we would like to mention the major branches of such work as possible extensions to our efforts.

Amnell and colleagues [4] consider code generation from timed automata [2] extended with tasks [35]. Timed automata can be seen as simple statecharts, without hierarchy, where the actions have durations and deadlines. A timed automaton is schedulable if it can be executed in a way that all the tasks always meet their deadline. Authors of [4] propose an algorithm that for a given schedulable timed automaton generates code, which guarantees that the deadlines are met.

Henzinger and colleagues [51] popularize a language and methodology

called Giotto, targeting mostly periodical programs for heavy data processing such as unattended helicopter control. Their compiler checks whether the model provided is actually schedulable and finds a schedule that guarantees the time safety.

3.6 Summary

In the above chapter we have introduced the problem of code generation from statecharts. We have sketched our requirements and described four major classes of attempts to solve this problem: the state-pattern method, the quantum framework, the interpretative methods and the BDD-based method. We have argued that the tools based on interpreters are most suitable for small constrained embedded systems without memory protection. This is the method used in our tool, SCOPE.

We have discussed the architecture of SCOPE, indicating available frontends, back-ends and internal model transformations. We have emphasized that SCOPE's architecture is modular and layered. SCOPE is easily retargetable. The generated C programs are portable.

The two main scientific contributions of this chapter are contained in section 3.4, which describes the model transformations: elimination of dynamic scopes and static conflict resolution. These algorithms (or rather properties that they ensure) are prerequisites for the code generation techniques described in the next chapter.

4

Back-End & Runtime

Our development is inevitably proceeding towards the back-end parts of the tool. We shall now focus on the actual essence of the code generation: translation of models to compact programs. We consider two main ways of performing this translation: one based on flattening the hierarchical models and one based on maintaining the hierarchy in the generated program.

Figure 4.1 recalls the structure of the entire tool, zooming into some details of the back-end part. The gray-shaded boxes represent stages of the code generator itself. Arrows represent data flow, or in other words internal representations of various stages. The input of the code generator is the output of the optimizer: a conflictless statechart annotated with static scopes. The core of the translation is performed in the first phase: the internal translator encodes the syntax of the model into data structures stored in intermediate arrays. The actual addresses and sizes of integer fields are not determined yet. This task is performed by the static data manager, once the entire structure is known. Then the code generator translates the intermediate arrays and the addressing/typing information into a collection of snippets of C abstract syntax. The ultimate output of the C pretty printer consists of several C files containing a ready to use control program. The last two stages are both standard and simple. We will focus on the internal translator mostly, just mentioning the important points of the static data manager.

Our presentation will occasionally rely on the use of pseudocode and diagrams, both abstracting away inessential details, but both precise enough to make the use of resources visible. For example stack allocation discipline will be presented as in the actual C implementation, and the diagrams will make it obvious how many bytes are used for essential data structures. Should you need to study the actual source code of implementation, please refer to the project website at [137]. We use the pseudocode both to present the algorithms of the code generator (implemented in Standard ML [95]) and of the runtime engine (a mixture of generated and hand-written C code).
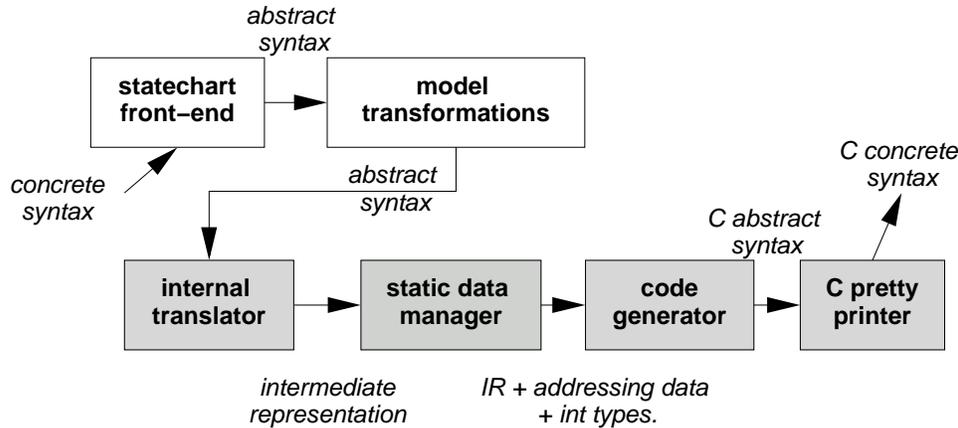
Figure 4.1: A structure of SCOPE's back-end implementation

We begin with a presentation of common principles of the runtime system (section 4.1), shared by both hierarchical and flat versions. In section 4.2 we introduce algorithms and data structures, specific for the hierarchical back-end. We evaluate them and measure against the stock code generator of IAR visualSTATE. In section 4.3 the specifics of the flat runtime are discussed. The remaining part of the chapter is devoted to the problem of *flattening*—translation of regular statecharts to flat ones, which can be efficiently represented and interpreted. First we asses the lower bound for complexity of a variant of this problem (section 4.4), then we relax the conditions slightly and propose an efficient flattening algorithm (section 4.5). We discuss the correctness of the algorithm and its efficiency. Finally we report the related work and conclude.

Sections 4.4 and section 4.5.5 can be safely skipped by more practically inclined readers. If you are only interested in what proves to be the most efficient code generation scheme described in this thesis, then only read sections 4.1, 4.3 and 4.5. The source code of hierarchical runtime engine (not the code generator itself) is presented in Appendix B, while an example of generated hierarchical model encoding can be found in Appendix E.3. Source code of the flat interpreter is in Appendix C, complemented by a flat encoding of example model in Appendix E.4.

4.1   Basics of the Runtime System

The two fundamental components of the synthesized program are the representation of current state and the reaction relation implementing the macrostep (see Fig. 4.2). Hardware synthesis techniques usually implement the current state using a feedback register which is modified by the combina-
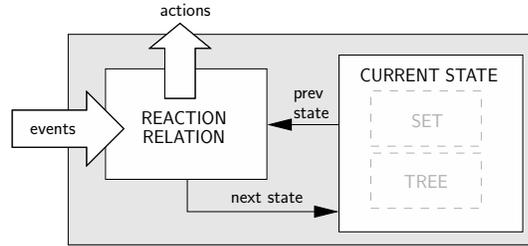
Figure 4.2: Typical structure of a synthesized program

tional block implementing the macrostep relation (see for example the work of Drusinsky in [31]). In software synthesis data structures for current state and reaction relation are needed. The so called *hierarchical code generation* uses an advanced data structure for representation of state, which allows a very direct implementation of the transition relation. The alternative way, called *flattening code generation*, embeds most structure of the state into the transition relation itself, while leaving the state representation fairly simple. SCOPE implements both methods.

In both cases the transitions are stored in a direct access table *tranidx* containing flat lists of transitions (see diagram on Fig. 4.3). Transitions triggered by event e belong to the *Trans*[e] list. Each transition is described by the number of positive conditions pc, the number of negative conditions nc, the lists of conditions themselves, a reference to a guard function and an action function[1], and finally the list of targets grouped by common scopes, if the runtime supports hierarchy. In the flattening variant the scopes are insignificant, so the targets are kept on a simple flat list.

When event e arrives, the respective list of transitions is interpreted by a microstep loop in a manner similar to Dijkstra's guarded commands. For each transition it is first checked that all positive states are active and all negative states are inactive. If this is the case then the transition is fired. It is skipped otherwise. Figure 4.4 presents a loop implementing a single microstep, assuming that e is the current event (compare to 2.63, p. 32). Another loop implements the macrostep relation (2.64-2.65, p. 32):

MACROSTEP()

    ▷ a global integer variable e stores current event
    while $e \neq$ NIL
        do *prev-conf* ← *next-conf*
           MICROSTEP()
           e ← DEQUEUE()

---

[1]Guard conditions can be reduced to flat lists of positive and negated literals, because the syntax (see 2.18, page 16) only allows negation at the variable level.
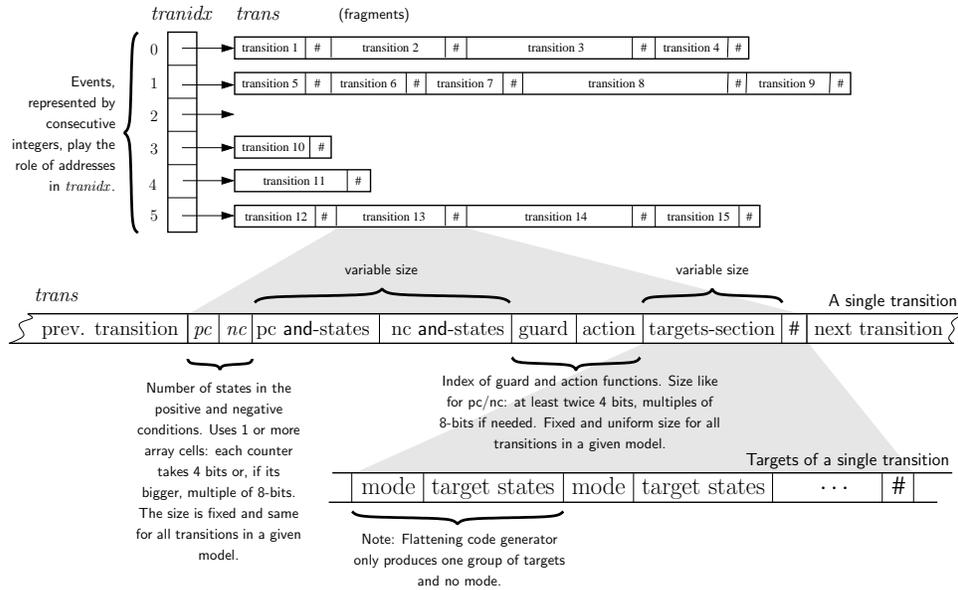
Figure 4.3: Direct access table storing transitions. All transitions reside consecutively in a single integer array *trans*. Pointers in *tranidx* are indexes to this array, not real pointers. The end of a given list is detected by comparison with the beginning of the subsequent list.

MICROSTEP calls the FIRE function presented below. It fires a transition if the guard is satisfied. Firing comprises exiting all the target scopes, executing the action function and entering all target states. The skeleton of the function is generic but relies on calls to a generated guard evaluator EVAL, an action executor EXEC as well as macros for decoding the transition fields, which are model dependent. The action executor EXEC also places local signals in the queue if needed.

FIRE(s : integer offset in *trans*)

    store action and guard references from *trans*[s] in *ac* and *gd*.
    if EVAL(gd) $\neq$ 0
        then exit all scopes on the targets list which begins at *trans*[s]
            EXEC(ac)
            enter each group of targets on *trans*[s]
    return

Variables *prev-conf* and *next-conf* represent the previous and the next state configuration. These are accessed and modified respecitvely in each microstep.

Functions ENQUEUE and DEQUEUE implement a signal queue using a global ringbuffer. ENQUEUE is called by the action executor whenever a

MICROSTEP()
> a global integer variable e stores an event identifier
> *tranidx* stores offsets in *trans*
$s \leftarrow$ *tranidx*$[e]$
next_tran: **while** $s \leq$ *tranidx*$[e+1]$
       do
             Store counter values from *trans*$[s]$ in *pc*, *nc*
             advance s to after *nc*
             > Verify positive part of the guard:
             $s' \leftarrow s + pc$
             while $s < s'$
                  do
                      if ACTIVE-AND(s)
                          then $s \leftarrow s + 1$
                          else advance s until # mark
                                  goto *next_trans*
             > Verify the negated part of the guard:
             $s' \leftarrow s + nc$
             while $s < s'$
                  do if ACTIVE-AND(s)
                          then advance s until # mark
                                  goto *next_trans*
                          else $s \leftarrow s + 1$
             FIRE(s)

Figure 4.4: An implementation of the macrostep relation.

signal needs to placed in the queue. Single pending signals are returned to the executor as results of the action calls. If a transition needs to trigger more than one signal then it returns a reference to a table of signals where unique groups that should be triggered together are stored.

Overflow safety for the signal queue is not automatically guaranteed. The user is obliged to provide the maximal length of the signal queue or its safe overapproximation. This bound can be obtained with knowledge of the model and good understanding of its works. Alternatively one can use visual-STATE model-checker to establish it. The model checker does not compute the bound on the signal queue size, but for a given bound can check if it is not violated. Subsequent runs of the model-checker for increasing sizes of the signal queue may be used to establish a suitable bound. One has to admit though that treatment of signals in the current version of visual-STATE's model checker (ver. 4.x-5.x) is far from satisfactory. The intensive use of signal queue (especially queues which are longer than one or two cells), causes an explosion of the reachable state space making the analysis highly inefficient.

The runtime representation has been designed with modest space requirements and fast access in mind. It is based on an observation that realistic models are relatively sparse: despite the multitude of attributes for states and transitions, developers hardly ever use all of them. Thus commonly used elements (initial markers, source states, targets) are implemented cheaply, whereas it is acceptable to use more space and access time for exotic ones: multiple targets, complicated conditions, exit/entry actions, history and multiple signals.

An initial marker, present once for each or-state, is an example of a commonly used element. Initial markers take no space in SCOPE's runtime representation. Instead children lists are reordered, so initial states become lists' heads.

A typical transition only uses a simple condition (a source state and a discrete event), an action and a single target state. These fields, stored in the static part of the transition record, are quickly accessible using fixed offsets. Multiple targets, a complex guard, and a transition scope are kept in the variable section of the record. Slower and more expensive field type indicators are used in this part, which is acceptable for rarely used elements.

Variables

According to the semantics of statecharts updates to variables should only be visible after the microstep step is completed. The assignment to x on a transition should not affect any value of x in other expressions evaluated within the same step. This problem is classically solved using double-buffering of variables, where the *lvalue* and the *rvalue* of a variable are separated. This is materially the very same technique that was described before for state

configurations (use of previous and next configurations). Two runtime variables are maintained for each model variable. One copy (rvalue) is used for reads, the other (lvalue) for write accesses. After the step is completed the lvalues are copied over the rvalues. If the number of variables is big, the cost of this operation may be significant. Also the size of writable memory employed increases. For this reason developers of highly constraint systems avoid double buffering, following the modeling style which is not prone to such subtleties.

Currently SCOPE does not implement double-buffering, although the extension would be straightforward. All experiments with visualSTATE (see section 4.2.4) have been performed with double buffering switched off to account for this lack.

Command and Expression Code

Despite the advances in optimization technology C compilers face hard problems caused by the type system and highly imperative semantics of the language. For instance, an automatic code generator is rather likely to produce redundant identical pieces of code, including complete function bodies. The C compiler must maintain all identical pieces to guarantee correctness of pointer comparisons (if function pointers are used). To avoid this identical pieces of code should not be generated to begin with. We implement a dynamic table of C code snippets which only saves fragments not seen before. We use it then to build the actual C program. This uniqueness detection uses a trivial syntactic criterion (identity), sufficient for automatically generated code and reasonable for user written code as we speak of short actions and expressions without local variables.

4.2   Hierarchical Back-End and Runtime

Hierarchical code generation keeps the resulting program as close as possible to the semantics of the model. Not only the behavior is preserved, but also the syntactical structure of the model is mimicked. Thus numerous syntactic objects of statecharts, like entry/exit actions, history states and notably the substate relation, are explicitly represented in the generated program and are present at runtime. This way it is easy to establish correctness of implementation. Also the linear relationship between the size of the model and the size of the generated program is obvious. Consequently a hierarchical code generator does not encode models in any complex way. The translation itself is straightforward, but the focus is on the design of efficient data structures for model representation.

The main difficulty of the hierarchical code generator is an efficient runtime representation, so that generated programs are not only asymptotically linear in the size of the model, but also the coefficient of the size function is
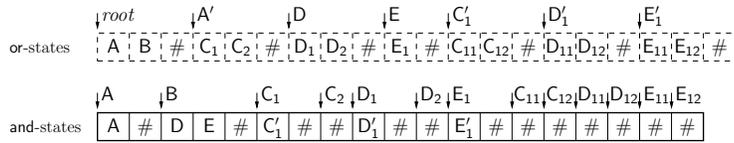
Figure 4.5: Hierarchy of fig.2.1b encoded in two arrays
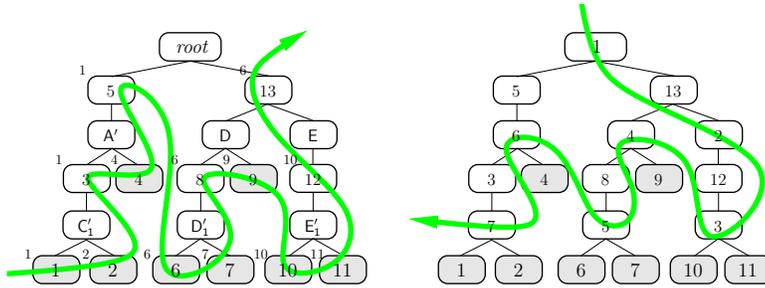
Figure 4.6: Labeling schemes for statechart hierarchy tree

low enough, so that the generated code remains competitive even for small models as often met in the industrial practice. The hierarchical code generator of SCOPE shows that this goal can be achieved. It performs reasonably on small and simple models. Results are especially good for bigger models, when it clearly wins with the industrial implementation based on flattening. One of the main reasons for which this became possible, is the elimination of dynamic scopes and conflicts, performed in the optimizer, allowing the removal of complicated machinery from the runtime interpreter.

4.2.1   Hierarchy Tree

The hierarchy tree is the essential data structure of hierarchical runtime representation. The tree itself is encoded in an integer array stored in a read-only memory. We exploit the regularity of state type alternation between and and or to recognize state type by its position in the tree, saving both space (no runtime type information) and time (no dynamic type-checks). Additionally and-states and or-states have separate name spaces, so identifiers are reused and become shorter. The two parts of the tree are saved in separate arrays. Figure 4.5 sketches an array representation for the tree of Fig. 2.1. All additional state attributes including parent information and entry/exit actions are omitted. The # marks denote endings of records.

In practice state addresses (array indexes) are used as state identifiers. If any of the two arrays is longer than the integer type sufficient to represent the number of states of a given type, state offsets double their size, which

```
              1      2  3      4      5      6      7
or-states   | 5 |13 |12 |10 |11 | 8 | 9 | 6 | 7 | 3 | 4 | 1 | 2 |-∞|

              1   2   3   4   5   6   7   8   9  10  11  12  13
and-states  | ∞ | ∞ | 7 | ∞ | 6 | ∞ | ∞ | 5 | ∞ | ∞ | ∞ | 3 | 4 | 2 | ∞ |
```
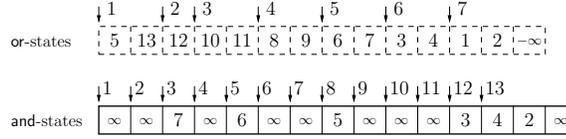
Figure 4.7: Array encoding of the tree on the right side of Fig. 4.6

would immediately affect the size of identifiers and hence all arrays. To defer this undesirable effect an intermediate dictionary (an array of offsets) is created in such case and states are addressed by an extra indirection at runtime. It can be shown that the space cost of dictionary is always smaller than the saving on the array size in such case. The decision whether this is needed, the actual generation of the indirection, and the assignment of fixed sized integer types to data, are the main tasks of the static data manager. Before that point all the analysis and generation uses large types internally for integer data and symbolic references for addressing.

Ancestorship queries are the most important operations on the hierarchy tree: they are performed whenever a state activity is checked. This happens when selecting transitions to fire and when selecting routes of activation for cross-level transitions or transitions targeting non-basic states. The visualSTATE statecharts require even more activity checks than UML state diagrams, since synchronization by states (using guards) is much more natural for them than synchronization by signals. A trivial implementation of hierarchical ancestorship check demands traversing the path between two states:

TRIVIAL-ACTIVE-AND($s : State_{\mathsf{and}}$)
    for $s' \in prev\text{-}conf$
        do if ANCESTOR(s, s')
            then return 1
    return 0

ANCESTOR($s : State, s' : State$)
    if $s' = $ NIL
        then return 0
        else return $s = s' \vee$ ANCESTOR($s, parent(s')$)

Unfortunately the most expensive case, when ancestorship does not hold, seems to be the most common one. To diminish the problem we propose a simple labeling scheme supporting checks based only on state labels.

Assume that and-states are numbered in a depth-first-search order (more precisely a postorder with left-to-right visiting of children). The identifier assigned to a given state s is greater than the identifier assigned to any of its descendants and all of them are greater than the identifier assigned to the leftmost descendant of s (see Fig. 4.6, left). For each and-state an interval of ancestorship identifiers can be computed. Then the left end-point of the interval (the left most descendant, or the LMD) needs to be saved for each and-state. The right end-point of the interval is the state identifier, which is always known when reaching the state. As such it does not need to be saved separately.

ACTIVE-AND-SE($s : State_{\mathsf{and}}$)

    for $s' \in prev\text{-}conf$

        do if $s' \geq LMD[s] \wedge s' \leq s$

            then return 1

    return 0

The labeling can be exploited even further to eliminate some record markers from the structure. Recall that and-states and or-states have separate name spaces, which means that they can be labeled in different ways. Note that the interval labeling of and-states does not necessarily demand for state identifiers to be consecutive numbers. Recall also that state identifiers are used as state pointers, which means that states are arranged in the same order in the arrays in which they are visited by the labeling algorithm.

It can be shown that if or-states are labeled in order dual to the one presented for and-states (DFS, preorder, with right to left visiting of children) both arrays exhibit an interesting property. Children lists in the or-state array form strictly increasing sequences of values, while sequences of children on lists in the and-state array are strictly decreasing (see Fig. 4.6, right). Moreover the monotonicity is always broken between lists of children of two neighboring states. This information can be used to distinguish record boundaries and instead of the # marks. Only if the state contains more attributes than just a children list, they are saved in front of the state record preceded by a mark: negative infinity for the array of and-states and positive infinity for array of or-states. A guarding mark should also be appended in the end of the arrays, and for basic states.

Figure 4.7 shows a representation of the tree shown on the right of Fig. 4.6. Additional fields have been suppressed and consecutive numbers instead of actual offsets were used to increase readability. The saving is especially visible in the or-states array, which by definition does not contain any leaves, and for deeper models with many internal nodes.

4.2.2   State Configuration Encoding

I have experimented with two simple state encoding techniques: a set-based
encoding and a flag-based encoding—a variant of classical 1-hot state assign-
ment adopted for software implementation of statecharts. The set encoding
is more compact, while the flag encoding yields faster programs. The set en-
coding relies heavily on optimized ancestor queries described in the previous
section. The performance of encodings is compared in section 4.2.4.

Set-Based Encoding

The set-based encoding maintains only a set of active basic states, as op-
posed to maintaining activity flags for all **and**-states. This is information is
sufficient to solve activity queries for all states.

We use static buffers, rather than dynamic data structures to represent
sets and queues. The set of active states is a simple buffer of elements with
empty cells in the end. The lack of gaps between elements is important
as we shall see that efficiency of state activity tests depends on the actual
number of elements filled in.

The implementation does not prevent overflows. To guarantee safety, a
bound on configuration size must be found statically. The exact maximum
size of the configuration can be computed using reachability analysis. For
fast compilations a cheaper estimation is needed. A simple recursive algo-
rithm is used to give an upper bound of the configuration size. The bound
for each basic **and**-state is assumed to be 1. The bound for each **or**-state is
the maximum of bounds for its children. The bound for each **and**-state is
the sum of children's bounds. Although very simple, this algorithm gives a
good improvement over the trivial bound—the number of all basic states.
Moreover the estimation is exact for purely sequential or entirely flat models:

CONFIGURATION-BOUND(s)

    if BASIC(s)
       then return 1
    elseif AND-STATE(s)
       then return $\sum_{s' \in children(s)}$ CONFIGURATION-BOUND(s')
    ▷ s is an **or**-state
    return $\max_{s' \in children(s)}$ CONFIGURATION-BOUND(s')

Interval labeling and set-based encoding can also be used for optimizing
some of the state exit operations. Observe that whenever a transition is fired
all states within its scope should be exited and then new states should be
activated. Standard exit procedure starts from active leaves of the respective
subtree and proceeds towards the top executing all exit actions on the way.

EXIT-AND(s)

    if $children(s) \neq \emptyset$

       then for $s' \in children(s)$

            do EXIT-OR(s')

         call $exit(s)$

         $next\text{-}conf \leftarrow next\text{-}conf \setminus \{s\}$

EXIT-OR(s)

    for $s' \in next\text{-}conf$

       do for $s'' \in children(s)$

            do if $LMD(s'') \leq s' \leq s''$

                then EXIT-OR(s'')

                   return

This can be improved for some states. An or-state (a transition scope) is said to be *exit-pure* if none of its descendants has any exit actions assigned. For such state another exit algorithm may be proposed. Instead of traversing the subtree and executing empty exit actions, one can scan the set of active states and simply delete all states between the end-points of the interval corresponding to the subtree.

EXIT-PURE-OR-SE(s)

    for $s' \in next\text{-}conf$

       do if $LMD(s) \leq s' \leq s$

           then $next\text{-}conf \leftarrow next\text{-}conf \setminus \{s'\}$

    return

Flag-Based Encoding

An alternative encoding for statechart configuration would preserve the information about all states, not only the basic states. The idea is to store identifier of active child for each or-state, or a distinct value if the state is inactive itself. This way activity checks become very efficient (and constant time) at the cost of updating the information for more states, whenever a transition fires. Also more writable memory is needed.

Hardware implementations of similar encodings [31] use $\lceil \log n \rceil$ bits for each or-state, where n is the number of its children. Access to subparts of machine word is relatively inefficient, when it comes to software implementations. A vector of cells with fixed size is used instead in SCOPE. The cell size should be sufficiently big to store the information for or-state node with the highest out-degree.

Model	*states*	*trans.*	*depth*	Executable Size [bytes]					
				VS	SC-SE	*ratio*	SC-FE	*ratio*	*ram*
actions01	4	1	3	3596	3752	1.04	3704	1.03	0
drusinsky89	19	14	7	3976	4192	1.05	4144	1.04	4
lift	18	19	3	4452	4432	1.00	4372	0.98	0
peer	275	192	23	12644	10352	0.82	10536	0.83	56
trios01	1121	840	9	28164	19848	0.70	24108	0.86	271
trios03	1121	840	9	60196	22048	0.37	24684	0.41	271

Table 4.1: Size results: IAR visualSTATE 4.3 vs SCOPE 0.11

Model	*states*	*trans.*	*depth*	Execution Time [s]				
				VS	SC-SE	*ratio*	SC-FE	*ratio*
actions01	4	1	3	7.61	6.27	0.82	6.02	0.79
drusinsky89	19	14	7	9.64	7.67	0.80	7.99	0.83
lift	18	19	3	15.66	30.33	1.94	21.30	1.36
peer	275	192	23	20.66	31.81	1.54	26.31	1.27
trios01	1121	840	9	541	730	1.35	255	0.47
trios03	1121	840	9	1139	751	0.66	260	0.23

Table 4.2: Speed results: IAR visualSTATE 4.3 vs SCOPE 0.11

4.2.3   Transitions

Another simple, but practically successful, optimization is the introduction
of several target types called *modes*. Most importantly we distinguish flat
and non-flat targets of transitions. Informally a target is *flat* if an arrow
drawn to it from the transition source does not cross any statechart levels:
it remains within the same or-state. Nonflat targets need to be decorated
by scopes as described previously. However, the computation of scope for
flat targets is very cheap and can be done by looking up the parent of the
source state in the hierarchy tree. Thus scope information does not need to
be saved for majority of transitions, bringing yet another space saving.

4.2.4   Evaluation

Let n be the number of all states in the model and t the number of tran-
sitions. The hierarchy tree and the transition table can be implemented in
$O(n)$ and $O(t)$ space respectively. The only point where the linearity can

be broken is the removal of dynamic scopes, which occasionally multiplies transitions. A transition can be multiplied at most $O(d^m)$ times, where d stands for depth of the tree and m is the maximum over number of targets on a single transition. Fortunately this term can be considered constant and is small in real models. The number of possible scopes is usually at most two or three. Also it is typical to have at most one dynamically scoped target on a transition while statically scoped targets do not cause any multiplication. Finally it is extremely uncommon to actually meet dynamically scoped transitions in real life models.

The elimination of end-of-state markers brings a constant saving of space in representation of the hierarchy. So does elimination of field indicators for commonly used elements (for instance initial states).

A single activity test in the set-based encoding costs $O(dn)$ time. SCOPE reduces this to $O(n)$ using descendants interval labeling. In practice n becomes the number of active basic states, which is much less than number of basic states. Flag-based encoding enjoys the constant time cost of a single activity test, while increasing the use of writable memory and administrative cost of firing. Still it seems that much more transitions are queried than fired at a single microstep, so the flag-based encoding is faster.

Experiments have been carried out both with SCOPE and IAR visualSTATE. Generated programs have been compiled with GCC 3.2 optimizing for size, on an x86 PC running Linux. Sizes are bare executables in bytes. Only the control algorithm and the runtime library were linked in. All references to external functions have been substituted with dummies. Running times are given in seconds, measured by triggering 10^7 random events, reinitializing the state machine before each event with probability of 0.01. The measuring was performed on a 450 MHz Pentium II.

Tables 4.1, 4.2 present selected results. The *states* column contains the total number of states (both and-states and or-states), *trans* shows the total number of transitions (excluding initial transitions), while *depth* gives the depth of hierarchy tree (counting both and-states and or-states). The minimal depth is 3, which is observed for flat models. *VS* denotes visualSTATE, *SC-SE* denotes SCOPE in state-based encoding mode, *SC-FE* denotes SCOPE using the flag-based encoding. Ratios are computed with respect to the visualSTATE measurements. The *ram* column presents the size of writable memory consumed additionally by the flag-based encoding comparing to the set-based encoding.

Similar experiments have been carried out using a nonoptimizing compiler (LCC for Linux) and optimizing embedded systems compilers from IAR Systems. The results were comparable; the only part of the program, that can be optimized by a C compiler is the runtime library. IAR compilers for PIC and AVR platforms shown that the visualSTATE's and SCOPE's runtime libraries differ by about 5% in size. Static integer tables encoding model data are beyond the scope of ordinary compiler optimizations and

thus they remain the same from platform to platform. I have reported GCC results since this is the most widely accessible reference platform.

Actions01 is a trivial example containing two basic states connected by a single transition. The size differences reflect the sizes of runtime engines. The hierarchical library seems to be only slightly bigger. The version of the library using flag-based encoding is smaller than the one for set-based encoding as the logics involved is much simpler.

Two other examples, *Drusinsky89* [29] and *lift*, show that the size of code produced by SCOPE is comparable to that of IAR IAR visualSTATE for small models. The difference seems to be acceptable. The latter of the two, *lift*, is a flat statechart (a set of concurrent state machines). It demonstrates the performance strength of flattening approach used in visualSTATE on flat models. The interpreter for the flat structure is very efficient. What slows it down is the growth of the structure itself, which is not observed for flat models.

A typical medium size model with irregular structure is represented by *peer*. The last two models, *trios01* and *trios03*, are highly concurrent and uniformly deep (the whole structure is equally deep). The latter one uses deep history on top level. Such models exhibit the size explosion problem of the flattening algorithm implemented in visualSTATE.

We should stress that the ratios presented in the tables above, computed for complete statically linked executables are slightly inexact. The model independent start-up and helper code generated by gcc, takes roughly 1kb in both cases distorting the output by roughly 1% in the model of size comparable with *peer*.

The overall result is that hierarchical generation technique seems to be feasible for small models, and scales well to large ones. Also if the cost incurred on writable memory is acceptable in a given application, the flag-based encoding should be used as this brings efficiency gains over the set-based approach. Nevertheless, as we shall see soon, the novel flattening algorithm proposed in this thesis, yields even more efficient code in general.

4.3   Flat Runtime

The main disadvantage of the hierarchical runtime is the size of the hierarchy tree representation and the complexity of operations (hierarchical activity checks and entering/exiting with tree traversals). It increases the writable memory consumption at runtime and makes it difficult for generic tools to automatically estimate the actual memory used and response times. The depth of recursive calls depends on the structure of the model, which in turn is encoded as data in integer arrays—an extremely hard case for automatic analysis.
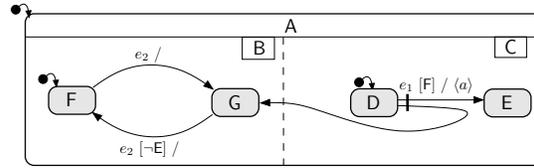
Figure 4.8: An example of a flat statechart

At the same time flat, concurrent communicating state machines, enjoy a very simple and efficient semantics. In this section we briefly present the runtime, which uses flat state machines as the model representation. In the following section we discuss problems involved in translating the original statechart model into this flat representation.

4.3.1 Flat Statecharts

A *Mealy machine* [90] is a finite state machine with transitions between states and sequences of atomic actions executed when a transition is fired. Each transition is labeled by a triggering event and a guard condition. A *flat statechart* (see figure 4.8) is a set of Mealy machines, operating concurrently in synchronous steps. The machines communicate by synchronization on active states.

Formally a flat statechart is a restriction of a hierarchical statechart. Flat statechart has a trivial hierarchy tree comprising four levels: (1) a *root*, (2) a single and-state, which contains (3) Mealy machines in or-states and (4) the basic states. Moreover exit and entry actions are not allowed: the *ex* function is a constant, always returning an empty sequence of outputs and the $\searrow$ relation forms a shallow tree: all and-states are basic states, except for the *root*.

Transitions are guarded by conditions on basic states and can only target basic states. The semantics of firing follows the individual scope scheme of the hierarchical semantics (section 2.2.6). It can be shown that in a flat statechart each target has an individual scope and all the scopes are flat (parents of the target state). Transitions never cause the exit of the *root* state. For example the transition sourced in the D state on Fig. 4.8, would cause a conflict under collective scopes semantics (UML, see p. 27): because it exits A and attempts to activate both D and E. In the flat (individual scopes) semantics it only activates G and E.

It can be shown that the individual scope's semantics and the requirement that transitions do not target the top most and-state, imply that the use of history is insignificant. States are never entered via history in such a setup. For this reason we also require the the history map *his* is empty. In fact in flat statecharts all states behave like history states—they never loose information on what state was active most recently.
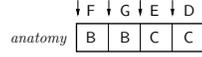
Figure 4.9: Anatomy of the flat statecharts of Fig. 4.8. As in the hierarchical back-end integers are used instead of state names. or-states and and-states enjoy separate namespaces.

Definition 4.1. *A statechart* $\mathcal{S} = (\Gamma_E, \Gamma_F, \Gamma_V, Signal, Var_E, Var_I, State, \searrow, ini, his, ex, en, Trans)$ *is called a flat statechart if:*

1. $\exists! s \in State_{\textbf{and}}.\ root \searrow s \wedge \forall s' \in State_{\textbf{and}}.\ s' \neq s \Rightarrow parent^2(s') = s$

2. $\forall s \in State_{\textbf{and}}.ex(s) = en(s) = \langle\rangle$

3. $\forall t \in Trans.\ \forall \sigma \in \Sigma.\ \forall s \in scope(t, \sigma).\ parent^2(s) = root$

4. $dom(his) = \emptyset$

We should stress that we have only defined flat statecharts as a restriction of hierarchical statecharts in order to avoid introduction of new language and its semantics. It is still worth to perceive flat statecharts intuitively as collections of synchronizing Mealy machines, because of their simple execution mechanism.

4.3.2 Implementation Details

The dynamic semantics of flat statecharts is an easily implementable subset of the hierarchical semantics. The hierarchy tree can be replaced by a simple map from state identifiers to state machines, which we shall call an *anatomy* of the flat statechart, to emphasize that this is not a fully fledged tree. If s is an and-state, then $anatomy[s] = parent(s)$. Fig. 4.9 presents the anatomy of the flat statechart of Fig. 4.8.

State configurations are maps from state machines (or-states) to active and-states, very much like in the flag-based encoding of the hierarchical back-end. The activity test for and-states is constant time:

Active-And-Flat(s)
    return *prev-conf*[*anatomy*[s]] = s

Finally, since all transitions are flat, they all have static scopes. This, together with lack of entry and exit actions, makes the entering process very simple:

Enter-And-Flat(s)
    next-conf[*anatomy*[s]] ← s
    return

The flat interpreter consists of three main functions MICROSTEP, MACROSTEP and FIRE as before. But the basic blocks used by FIRE are much simpler. Exiting states can simply be ignored. There is no exit actions and entry automatically performs exit via assignment in the state vector. Entering is constant time as we have seen above, and requires no recursive traversals. Consequently flat statecharts can be implemented using a simpler runtime system and less mutable data structures, which results in reduced consumption of writable memory. Our flat interpreter uses tiny amounts of writable memory: an order of ten integers plus the size of the current configuration vector and a very shallow bounded call stack—a significant advantage over the hierarchical version.

4.4   Lower Bound for Flattening

In the previous section we have appreciated the simplicity of our implementation of flat statecharts. In order to exploit this simplicity, one needs a translation algorithm from hierarchical statecharts to flat statecharts. We call this transformation *flattening*. Flattening is widely applied to hierarchical models both in theoretical and practical settings. It has been used to give the semantics of hierarchical languages [45] and to provide algorithms for code generation [57], automatic testing [12], and model checking [26]. Not only flat models can be easily interpreted with very limited writable memory usage, but they are also easier to analyze for worst-case execution time. Flat models can be more easily translated to hardware circuits. This makes flattening specifically attractive for code generation targeting constraint embedded systems. In the following sections we should study the transformation of flattening from two angles. First, a formal statement of lower-bound on model size increase shall be given. Then we shall relax the problem definition, to the point, where an efficient algorithm can be proposed. We describe its implementation and evaluate it.

Literature mentions a multitude of meanings for flattening and related concepts. Let me stress that the meaning given above is different from generation of a single product machine for all concurrent components. Our understanding of hierarchy, concurrency and flattening is rather similar to that of [3, 5, 119, 26] and substantially different than that of [16, 113, 118].

We should use a relativized bisimulation as a correctness criterion. The flattening algorithm, which we will ultimately propose, relies on the fact that the environment does not distinguish equivalent interleavings of concurrent activities in the statechart.

Definition 4.2. *Statechart $\mathcal{S}_1$ simulates statechart $\mathcal{S}_2$, written $\mathcal{S}_1 \leqslant \mathcal{S}_2$, if each macrosteps of $\mathcal{S}_1$ can be mimicked by a macrosteps $\mathcal{S}_2$ and both macrosteps advance models to state configurations where they still simulate*

each other:[2]

whenever $\langle \sigma_1, \varrho_1, \eta_1 \rangle \xrightarrow{e \quad os!} \langle \sigma'_1, \varrho'_1, \eta'_1 \rangle$

then also exist σ'_2, ϱ'_2 and η'_2 such that $\langle \sigma_2, \varrho_2, \eta_2 \rangle \xrightarrow{e \quad os!} \langle \sigma'_2, \varrho'_2, \eta'_2 \rangle$

and the same property holds for primed global states.

Definition 4.3 (Flattening). *Let F be an algorithm translating hierarchical statecharts to flat statecharts. F is a flattening algorithm if for any hierarchical statechart $\mathcal{S}$ it yields a flat statechart $\mathcal{S}'$ such that $\mathcal{S}' \leqslant \mathcal{S}$.*

Note that the simulation requirement is not trivial for input-enabled systems. In particular an algorithm always returning an empty statechart containing just the *root* state is not a flattening algorithm, because traces of the empty statechart contain empty outputs and us such are not legal traces of any nonempty statechart producing outputs.

The following theorem states a lower bound for the flattening problem: flattening cannot be achieved in polynomial space if the target statecharts is restricted only to communication via guards and communication via signals is disallowed.

Theorem 4.4. *There exists a hierarchical statechart $\mathcal{S}$ not using signals, such that for any flat statechart $\mathcal{S}'$ such that $\mathcal{S}'$ does not use signals and $\mathcal{S}' \leqslant \mathcal{S}$:*

1. *The size of $\mathcal{S}'$ is in $\Omega(2^{\sqrt{s}})$, where s is the size of $\mathcal{S}$.*

2. *Previous claim holds even if $\mathcal{S}$ is restricted to binary inputs and outputs.*

3. *The lower bound with growth rate arbitrarily close to the exponential, can be constructed by choosing $\mathcal{S}$ with sufficient amount of concurrency.*

Note that the second claim of the above theorem is stronger than the initial one. It says that the lower bound holds even for a subset of hierarchical statecharts over binary alphabet (decreasing the set from which $\mathcal{S}$ can be chosen). So the first claim is a special case of the second claim. The third claim is even stronger saying that the lower bound can be increased arbitrary close to the exponential function. We will show how to construct $\mathcal{S}$ so that the degree of the root in the exponent approaches one, as the amount of concurrency in $\mathcal{S}$ increases.

---

[2]A formal definition based on fixpoint theory will be given in chapter 5.
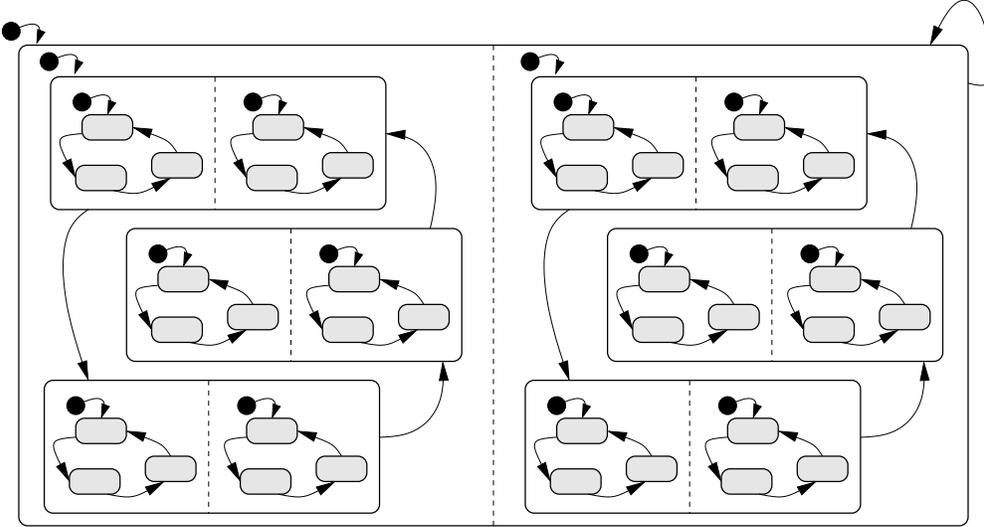
Figure 4.10: (2,3)-model of and-depth 3, also a (2,3)-model of 58 states

4.4.1   Proof

The proof proceeds by identifying an infinite family of (α, β)-models such that each of the members in the family has a superpolynomial reachable state space and each state configuration yields a different sequence of exit outputs. Such sequences cannot be represented in any flat model without equivalent multiplication of transitions. Finally we show that the family of (α, β)-models contains statecharts for which hardness of flattening problem is arbitrary close to an exponential of model size.

In the proof we shall use the following auxiliary notions. The *out-degree* of a state is the number of its children (i.e. the out-degree of the node in the hierarchy tree). The *depth* of the model, denoted d, is the number of states in the longest path in the hierarchy tree leading from *root* to a leaf. All models always have even depth (as the *root* is an or-state and so are all leaves). A variant of depth—called and-*depth*, denoted $\hat{d}$, only reflects number of and-states in the paths: $\hat{d} = \frac{d}{2}$. All flat statecharts have depth $d = 4$ and $\hat{d} = 2$.

We will denote the number of all states in the model by $n = |State|$. Finally the size of the model is defined as the size of all its guards, actions output sequences and the number of states.

Family of (α, β)-models

Consider a family of statecharts with fixed out-degree $\alpha \geq 2$ for nonbasic and-states and fixed out-degree $\beta \geq 2$ for not-*root* or-states. Each and-state has a unique exit action assigned. For each and-state in the model there is a transition sourced in that state. Each transition has a unique event triggering it. The targets of transitions are selected in such a way that there is a cycle over and-states in any particular state machine at any level. Note that this way every statically legal configuration in the statechart is reachable.
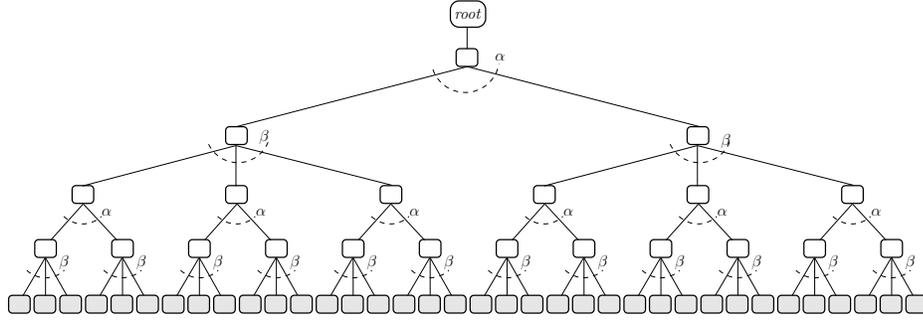
Figure 4.11: Hierarchy tree of (2,3)-model of figure 4.10

We will indicate a specific model in the family by giving its parameters and size, calling it an (α, β)-model of and-depth $\hat{d}$ or an (α, β)-model of n states. In the latter case n has to be consistent with α and β. Figure 4.10 presents a (2,3)-model of and-depth 3.

Note that the size of any model depends on size of actions and guards, the number of transitions and number of states. The number of actions and transitions in a given (α, β)-model is the same as the number of states. Thus, from now on, we will use the number of state n as a measure over (α, β)-models instead of the more general size s.

Reachable State Space

Let $width_{\hat{k}}$ denote the number of states on and-depth $\hat{k}$ (i.e. on $\hat{k}$th level of and-states) in an (α, β)-model:

$$width_{\hat{k}}^{(\alpha,\beta)} = (\alpha\beta)^{\hat{k}-1} \tag{4.1}$$

In particular $width_{\hat{d}}$ denotes the number of basic states in a given (α, β)-model. The number of active states of an (α, β)-model at and-depth $\hat{k}$ is given by:

$$active_{\hat{k}}^{(\alpha,\beta)} = \alpha^{\hat{k}-1} \tag{4.2}$$

The total number of states as a function of and-depth can be described with the following recurrence:

$$n_1^{(\alpha,\beta)} = 2$$
$$n_{\hat{d}}^{(\alpha,\beta)} = n_{\hat{d}-1}^{(\alpha,\beta)} + width_{\hat{d}-1}^{(\alpha,\beta)} \cdot (\alpha + \alpha\beta). \tag{4.3}$$

The recurrence, solved and inverted, gives the and-depth of the model as a function of the number of states n:

$$\hat{d}^{(\alpha,\beta)} = \log_{\alpha\beta}\left[\frac{\beta}{\beta+1}(n-1)(\alpha\beta-1) + \frac{\beta(\alpha+1)}{\beta+1}\right] \tag{4.4}$$

for legal combinations of values of α, β and n. Formula (4.4) gives a translation from functions over and-depth to functions over model size.

Recall that all statically legal configurations are reachable in (α, β)-models, so the number of reachable states in a model of depth $\hat{k}$ given by $R_{\hat{k}}^{(\alpha,\beta)}$, is equal to the number of possibilities in which active sets of states can be selected according to semantics of statecharts.

$$R_1^{(\alpha,\beta)} = 1 \tag{4.5}$$

$$R_{\hat{k}}^{(\alpha,\beta)} = \sum_{i=1}^{R_{\hat{k}-1}^{(\alpha,\beta)}} (\beta^\alpha)^{active(\hat{k}-1)} \tag{4.6}$$

The term under summation is independent of sum index, because of the high regularity of the model. Each configuration at level $\hat{k} - 1$ can be refined to exactly the same number of configurations on level $\hat{k}$. Thus the recurrence simplifies to:

$$R_1^{(\alpha,\beta)} = 1 \tag{4.7}$$

$$R_{\hat{k}}^{(\alpha,\beta)} = R_{\hat{k}-1}^{(\alpha,\beta)} \beta^{\,\alpha active(\hat{k}-1)}, \tag{4.8}$$

which can be solved giving:

$$R_{\hat{d}}^{(\alpha,\beta)} = \beta^{\frac{\alpha^{\hat{d}}-\alpha}{\alpha-1}} \tag{4.9}$$

This shows that the size of the reachable state space is double exponential in the depth of the (α, β)-model. Substitute (4.4) to obtain the size of reachable state space as a function of model size:

$$R_n^{(\alpha,\beta)} = \beta^{\frac{\left[\frac{\beta}{\beta+1}(n-1)(\alpha\beta-1)+\frac{\beta(\alpha+1)}{\beta+1}\right]^{\log_{\alpha\beta}\alpha}-\alpha}{\alpha-1}}, \tag{4.10}$$

for any fixed choice of α and β. The $R_n^{(\alpha,\beta)}$ function is $\Omega(2^{n^{\log_{\alpha\beta}\alpha}})$. Moreover if $\alpha = \beta$ then $R_n^{(\alpha,\alpha)} \in \Omega(2^{\sqrt{n}})$. The size of reachable state space for (α, β)-models is double exponential in the model depth, but superpolynomial and subexponential in the model size.

Succinctness

Let us deal with the unlimited alphabet case first, when arbitrary many input and output symbols are allowed in our statecharts. We shall see that (α, β)-models cannot be translated to flat statecharts without superpolynomial growth of size. Consider the top level loop transition in any of (α, β)-models. This transition may be enabled in any reachable configuration. Each configuration yields a unique exit output (one of possible interleavings) as each state has a unique exit action assigned. Thus the top transition may produce $R_{\hat{d}}^{(\alpha,\beta)}$ distinguishable outputs. When flattening this transition, $R_{\hat{d}}^{(\alpha,\beta)}$ output reactions need to be expressed. As many other sequences are illegal, the only way to guarantee a sequence of actions to be generated in the flat models in a fixed order is to place outputs on the same transition. Otherwise, if the parts were split across various transitions, no guarantee can be given in which order outputs will be generated. Thus at least as many transitions as reachable state configurations are needed in this cases. Take $\alpha = \beta$ and the first claim of theorem 4.6 is reached.
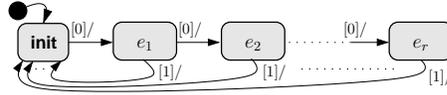
Figure 4.12: An extra component decoding the binary input

Input-Output Alphabet

In order to prove the second claim of theorem 4.6, namely that the same lower bound holds for binary input output alphabets, it suffices to show a polynomial translation of statecharts over arbitrary alphabet to statechart over binary alphabet. The translation should preserve the semantics of original statechart allowing triggering non binary events by encoding in binary sequences and similar observation of non-binary effects with distinguishable binary words.

Input encoding. We assume a finite number of events in input alphabet, indexed from 1 to r. We will use $i + 1$ bits to encode the firing of event e_i. First i *zero* symbols are sent, followed by a single *one* symbol. The translated model continues to receive *zero* symbols, advancing the counter of arriving event, and fires relevant transitions, when *one* arrives. A fresh component, illustrated on figure 4.12, is added to the model being translated by means of concurrent composition.

Then the triggering event on every transition in the old model is changed to 1. If the original transition was fired by event e_i then an extra term is conjuncted to transition's guard enforcing that state e_i is active. A transition $s_1 \xrightarrow{e_i\ [g]/os} s_2$ becomes $s_1 \xrightarrow{1\ [g \wedge e_i]/os} s_2$. The size of each transition has been increased by a constant factor.

The resulting model operates over binary input symbols, still presenting the same behavior and properties, modulo encoding. Moreover the size of the new model is linear in the number of transitions in the original model. In the worst case as many new states and new transitions have been added as there were transitions in the original model (if each transition was fired by unique event).

Output encoding. The translation from models over arbitrary output alphabet to models over binary output alphabet is even easier. It suffices to use any isomorphic encoding of natural numbers in binary alphabet and instead of every output generate a corresponding sequence of binary outputs.

The above encodings are generally useful whenever complexity proofs for statecharts need to be generalized to models over binary alphabets. In our case we notice that (α, β)-models can be translated within polynomial bounds to a corresponding family over binary alphabets. The whole proof can be rephrased in this framework – the properties of models are not changed. All configurations are reachable and each configuration gives rise

to a unique set of exit sequences. One still obtains the same superpolynomial order of growth, which finishes the proof of the second claim of theorem 4.6.

Improving the Lower Bound

Let us return to the lower bound on the number of configurations $\Omega(2^{n^{\log_{\alpha\beta}\alpha}})$. Note that the innermost exponent in the lower bound function is a constant from the interval $(0;1)$. Moreover if one extends the amount of concurrency in the model (controlled by α), keeping the amount of sequentiality (controlled by β) constant, the exponent approaches 1. Thus one can give a lower bound of the size being arbitrarily close to exponential in the sense of growth rate.

This shows the third claim of theorem 4.6. It is harder to flatten more concurrent models, despite the fact that concurrency is preserved by flattening. Hierarchy is strengthened by concurrency. □

The proof naturally suggests an algorithm of the same asymptotic complexity for flattening of statecharts. Thus for our (α, β)-models, which are a kind of regular statecharts (they have regular tree structure) the bound given is tight:

Observation 4.5. *Any (α, β)-model of n states can be flattened to a statechart with size in $\Theta(2^{n^{\log_{\alpha\beta}\alpha}})$.*

This also means that for regular statecharts in this sense an exponential limit on the lower bound cannot possibly be reached (which does not mean that the bound is tight for statecharts in general).

4.5 Polynomial Flattening

As we have seen in the previous section it is impossible to flatten hierarchical statecharts efficiently without using signals. We shall now demonstrate how signals and signal queue become instrumental in designing an efficient flattening algorithm. This result is relatively surprising as it contradicts a widely held but informal belief that a polynomial solution for this problem does not exist. At the same time experiments show that the quality of resulting code exceeds the quality of programs synthesized by the hierarchical code generator described in section 4.2.

Theorem 4.6. *For any hierarchical statechart $\mathcal{S}$ there exists a flat statechart $\mathcal{S}'$ such that $\mathcal{S}' \leqslant \mathcal{S}$ and the size of $\mathcal{S}'$ is at most polynomial in the size of $\mathcal{S}$.*

In fact this result holds even if the set of internal signals of $\mathcal{S}'$ is restricted to two distinctive values (note that this is a restriction on the target, not the source language, which would be trivial). This can be concluded using

binary encoding techniques presented in the previous section and shall not
be discussed further here.

We shall present the flattening algorithm in a declarative style, as a
syntax-driven transformation of statechart elements. Consider a hierarchi-
cal statechart: $\mathcal{S} = (\Gamma_E, \Gamma_F, \Gamma_V, \mathit{Signal}, \mathit{Var}_E, \mathit{Var}_I, \mathit{State}, \searrow, \mathit{ini}, \mathit{his}, \mathit{ex}, \mathit{en},$
$\mathit{Trans})$. We show how to construct a flat statechart $\mathcal{S}' = (\Gamma_E, \Gamma_F, \Gamma_V,$
$\mathit{Signal}', \mathit{Var}_E, \mathit{Var}_I, \mathit{State}', \searrow', \mathit{ini}', \mathit{his}', \mathit{ex}', \mathit{en}', \mathit{Trans})$ such that $\mathcal{S}' \leqslant \mathcal{S}$.

Observe that the environment events of $\mathcal{S}'$ are the same as events of $\mathcal{S}$.
The maps ex', en' and his' are empty. The interface to functions Γ_F, variable
sets Var_I, Var_E and types Γ_V remain unchanged. In fact the entire part of
each model not directly related to hierarchy will remain unmodified in the
flat model. Thus in the description of the algorithm below, we occasionally
take the freedom to ignore presence of variables and expressions in conditions
and actions of transitions. It is straightforward to incorporate them though.

4.5.1   The Algorithm

The or-states of $\mathcal{S}'$ are mostly identical with the or-states of $\mathcal{S}$. We add two
additional states: I_{or} used to implement administrative internal rules, and
$\mathit{root}_{\mathsf{or}}$, which will be the new top level state.

$$\mathit{State}'_{\mathsf{or}} = \mathit{State}_{\mathsf{or}} \cup \{\mathit{root}_{\mathsf{or}}, I_{\mathsf{or}}\} \tag{4.11}$$

The set of and-states is extended by an administrative state I_{and} and a top
level and-state $\mathit{root}_{\mathsf{and}}$ (the only child of $\mathit{root}_{\mathsf{or}}$):

$$\mathit{State}'_{\mathsf{and}} = \mathit{State}_{\mathsf{and}} \cup \{\mathit{root}_{\mathsf{and}}, I_{\mathsf{and}}\} \tag{4.12}$$

The new substate relation $\searrow'$ is a sliced version of the old one. Relationships
between or-states and and-states are kept, but all the or-states get a common
parent $\mathit{root}_{\mathsf{and}}$, which is a direct descendant of $\mathit{root}_{\mathsf{or}}$:

$$I_{\mathsf{or}} \searrow' I_{\mathsf{and}} \land \mathit{root}_{\mathsf{or}} \searrow' \mathit{root}_{\mathsf{and}} \land \forall s \in \mathit{State}_{\mathsf{and}}.\ \mathit{parent}(s) \searrow' s \tag{4.13}$$

$$\mathit{root}_{\mathsf{and}} \searrow' I_{\mathsf{or}} \land \forall s \in \mathit{State}_{\mathsf{or}}.\ \mathit{root}_{\mathsf{and}} \searrow' s\ , \tag{4.14}$$

where *parent* is defined based on the substate relation $\searrow$ of $\mathcal{S}$. The new hi-
erarchy tree (anatomy) created by flattening the tree of Fig. 2.1 is presented
on Fig. 4.13.

The new initial marking ini' is obtained by extending the original mark-
ing with extraneous states I_{and} and $\mathit{root}_{\mathsf{and}}$—both the only, hence trivially
initial, states in their state machines:

$$\mathit{ini}'(s) : \mathit{State}'_{\mathsf{or}} \to \mathit{State}'_{\mathsf{and}} = \begin{cases} \mathit{ini}(s) & \text{if } s \in \mathit{State}_{\mathsf{or}} \\ \mathit{root}_{\mathsf{and}} & \text{if } s = \mathit{root}_{\mathsf{or}} \\ I_{\mathsf{and}} & \text{if } s = I_{\mathsf{or}} \end{cases} \tag{4.15}$$

Clearly the size of $\mathit{State}' = \mathit{State}'_{\mathsf{or}} \cup \mathit{State}'_{\mathsf{and}}$ is linear in the size of the
original State set $\Theta(|\mathit{State}|)$.
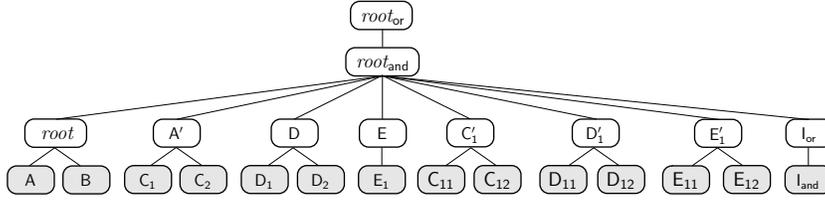
Figure 4.13: The anatomy resulting after flattening the tree of Fig. 2.1

Guards

Guards are flattened by computing ancestor closures over states of $\mathcal{S}$, so that invariants of the hierarchical configurations are enforced on the flat configurations as well, but this time at the transition level.

$$
flat(g) = \begin{cases}
true & \text{if } g = true \\
\neg flat(g_1) & \text{if } g = \neg g_1 \\
flat(g_1) \wedge flat(g_2) & \text{if } g = g_1 \wedge g_2 \\
\bigwedge_{p \in P} p & \text{if } g = s, \ s \in State_{\mathsf{and}} \\
& \quad \text{and } P = ancest^*(s) \setminus State_{\mathsf{or}} \ ,
\end{cases}
\tag{4.16}
$$

where $ancest^*$ is defined based on substate relation $\searrow$ of $\mathcal{S}$. For example the guard $D_1 \wedge D_{12} \wedge \neg E_{11}$ is flattened to:

$$
flat(D_1 \wedge D_{12} \wedge \neg E_{11}) = D_{12} \wedge D_1 \wedge B \wedge \neg(E_{11} \wedge E_1 \wedge B)
\tag{4.17}
$$

At this point the guards are suitable for placing in statecharts with unrestricted guard syntax, for example UML. In visualSTATE the syntax of guards is limited (see grammar at 2.18). It does not allow negation of arbitrary expressions, but just for literals. For this reason the result of flattening needs to be expanded usign distributive laws to a DNF formula:

$$
(D_{12} \wedge D_1 \wedge B \wedge \neg E_{11}) \vee (D_{12} \wedge D_1 \wedge B \wedge \neg E_1) \vee (D_{12} \wedge D_1 \wedge B \wedge \neg B) \tag{4.18}
$$

Unsatisfiable clauses are removed in practice (see the last one above) and then the transition gets multiplied, with each conjunctive clause being a guard on one copy. This expansion gives a potentially exponential growth of the number of transitions in the number of negations in the original guard. As we shall see later, this exponential growth does not affect the performance of the algorithm in practice. Transition guards are typically short and do not contain many negations. One can safely assume that the number of negated references does not exceed five in realistic models. So in practice the number of transitions only grows polynomially and each of the transitions is only polynomially bigger than originally.

Action Transitions

Entry and exit actions are not available in the flat target language. We shall implement them by generating an *action transition* for each of them. For each and-state s, define the following transitions:

$$t_s^{\text{ex}} = I_{\text{and}} \xrightarrow{e_s^{\text{ex}} \ [flat(s)]/ex(s)} I_{\text{and}} \qquad t_s^{\text{en}} = I_{\text{and}} \xrightarrow{e_s^{\text{en}} \ [flat(parent^2(s))]/en(s)} s, \qquad (4.19)$$

where e_s^{ex} and e_s^{en} denote fresh signals not belonging to $Event \cup Signal \cup Action$. These signals may now be used to trigger entry and exit actions. Note that the exit transition will fire if s is active, while the entry transition only activates s if the nearest and-state ancestor is active, so that the invariant for state configurations is preserved. The action transitions for state C_1 of Fig. 2.1 are:

$$I_{\text{and}} \xrightarrow{e_{C_1}^{\text{ex}} \ [C_1 \wedge A]/release()} I_{\text{and}} \text{ and } I_{\text{and}} \xrightarrow{e_{C_1}^{\text{en}} \ [A]/reserve()} C_1 \ . \qquad (4.20)$$

Similarly an action transition is generated for each hierarchical transition t:

$$t_t = I_{\text{and}} \xrightarrow{e_t \ [true]/os_t} I_{\text{and}} \ , \qquad (4.21)$$

where e_t is a fresh signal. This signal may now be used to trigger the action of the original hierarchical transition. An action transition corresponding to transition t_1 in Fig.2.1 is:

$$I_{\text{and}} \xrightarrow{e_{t_1} \ [true]/\langle o_1, s_1 \rangle} I_{\text{and}} \ . \qquad (4.22)$$

Later on we shall schedule action transitions in the proper sequences to implement traces of original statechart. Note that so far we have added a number of transitions linear in the number of states and transitions in the original model.

Interface

The input and output alphabets cannot be changed, otherwise the flat statechart would violate the implementation condition trivially. The set of internal signals is extended with the administrative signals mentioned before plus a new one for each history state:

$$Signal' = Signal \cup \{e_s^{\text{en}} | s \in State_{\text{and}}\} \cup \{e_s^{\text{ex}} | s \in State_{\text{and}}\}$$
$$\cup \{e_t | t \in Trans\} \cup \{e_s^{\text{h}} | s \text{ is a history state}\} \ . \quad (4.23)$$

The size of $Signal'$ is linear in the size of the original model.
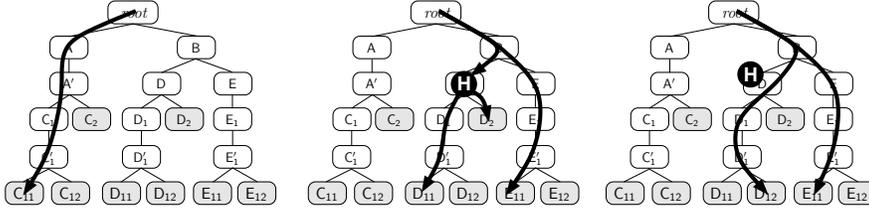
Figure 4.14: Entry schedules for t_1 (left), t_2 (middle) and t_3 (right). The path of t_1 is static, t_2 relies on dynamic history choice, and t_3 circumvents history with explicit guiding targets.

Entry Schedule

Firing a hierarchical transitions has three phases: exit the scope, execute actions, and enter the scope. We realize the entering and exiting phases by generating signal *schedules*. Schedules are sequences of administrative signals, which when interpreted by a microstep iteration realize the semantics of the original hierarchical transition.

An *entry schedule* has two parts: a statically computable part and a dynamic, history-dependent, part. The static part can be determined at compile time by computing a closure of the *ini* function guided by the set of explicit targets. The computation of the schedule continues until a history state is reached (see Fig. 4.14), where it forks to account for all variants in which entering may proceed from that state. The variants are guarded on the last value of history, so that only one will execute at each firing.

Two mutually recursive functions ENTRY-OR and ENTRY-AND realize the hierarchy traversal guided by the set of goal states *ts*. They stop at a history state or a basic state. They generate signals firing the entry transitions of respective states and a history transition if needed. The history state can only be bypassed if the targets below it are explicitly specified (see the example of t_3 on Fig. 4.14). Otherwise the function follows all possible history versions, relying on the guards on ancestor activity encoded in entry transitions to enforce the right path at runtime.

fun ENTRY-OR($s : State_{\mathsf{or}}, ts : State^*_{\mathsf{and}}) : Signal'^*$

    let $\{s_1, \ldots, s_p\} = children(s)$
    in **if** $s \in dom(his)$
        then $\langle e_s^{en}, e_s^{h} \rangle \hat{\ } \text{ENTRY-AND}(s_1, ts) \hat{\ } \ldots \hat{\ } \text{ENTRY-AND}(s_p, ts)$
        else $\langle e_{\text{DEFAULT}(s,ts)}^{en} \rangle \hat{\ } \text{ENTRY-AND}(\text{DEFAULT}(s, ts), ts)$

fun ENTRY-AND($s : State_{\mathsf{or}}, ts : State^*_{\mathsf{and}}) : Signal'^*$

    let $\{s_1, \ldots, s_p\} = children(s)$
    in ENTRY-OR($s_1, ts) \hat{\ } \ldots \hat{\ } \text{ENTRY-OR}(s_p, ts)$

fun $\textsc{Default}(s : State_{\text{or}}, ts : State^*_{\text{and}}) : State_{\text{and}}$
   if $\exists t \in ts, p \in children(s).\ p \searrow^*_{\searrow} t$
      then p
      else $ini(s)$

The semantics does not specify in which order concurrent components should be entered by $\textsc{Entry-And}$. Any order consistent with $\searrow$, including interleaved entry traces of concurrent components, is legal. The value returned by $\textsc{Entry-And}$ corresponds to a single deterministic choice of such a sequence. The omission of other choices in interleaving is permitted because our implementation relation is based on simulation.

The $\textsc{Default}$ function is a helper, which determines a default child for a given or-state. It checks whether further targets are specified below and follows the indicated path if available. The entering path below the history states cannot be decided at compile time, as the actual entry schedule depends on the runtime properties of these states. An administrative history signal e^h_s is triggered and a *history transition* is added for each child p of history state s, so that only one child of the history state is entered at a time.

$$I_{\text{and}} \xrightarrow{e^{\text{h}}_s\ [flat(p)]/en(p)} I_{\text{and}} \tag{4.24}$$

These history transitions are similar to entry transitions, except that they have stronger firing conditions. The guard $flat(p)$ guarantees that only one of these transitions will fire whenever e^h_s is triggered—the one entering the most recently active child—which corresponds to a runtime choice of entry schedule. After e^h_s entry schedules for all children of children of s are appended. Due to the guard on parent activity in entry transitions (4.19) only one cascade will actually have effect at runtime.

An entry schedule of a transition is computed starting with its scope and the set of targets. The schedule of t_2 is: $\langle e^{\text{en}}_{\text{B}}, e^{\text{h}}_{\text{D}}, e^{\text{en}}_{\text{D}_{11}}, e^{\text{en}}_{\text{E}_1}, e^{\text{en}}_{\text{E}_{11}} \rangle$. The schedule of t_3 is: $\langle e^{\text{en}}_{\text{B}}, e^{\text{en}}_{\text{D}_1}, e^{\text{en}}_{\text{D}_{12}}, e^{\text{en}}_{\text{E}_1}, e^{\text{en}}_{\text{E}_{11}} \rangle$, see Fig. 4.14.

Exit Schedule

The *exit schedules* are much easier to compute than the entry schedules. For a given scope s we define $\textsc{Exit}(s)$ to return the sequence of exit signals for and-state descendants of s produced in postorder traversal. For instance $\textsc{Exit}(A) = \langle e^{\text{ex}}_{\text{C}_{11}}, e^{\text{ex}}_{\text{C}_{12}}, e^{\text{ex}}_{\text{C}_1}, e^{\text{ex}}_{\text{C}_2}, e^{\text{ex}}_{\text{A}} \rangle$. Note that any entry or exit schedule cannot be longer than $|State_{\text{and}}|$.

Transition Schedule

Each hierarchical transition $t = s \xrightarrow{e\ [g\ exp]/os} ts$ is translated to a flat transition t' which schedules the relevant signals realizing the semantics. The condition
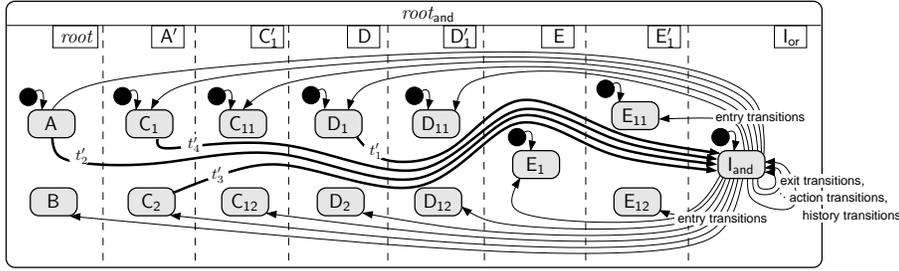
Figure 4.15: An imprecise but intuitive overview of results of flattening of statechart of Fig. 2.1

part of the transition remains unchanged, except for the flattened guard:

$$t' = s \xrightarrow{e \ [\mathit{flat}(s \wedge g) \ \mathit{exp}]/\textsc{Exit}(\mathit{scope}(t)) \hat{} \ \langle e_t \rangle \hat{} \ \textsc{Entry-Or}(\mathit{scope}(t),\, ts)} I_{\mathsf{and}} \qquad (4.25)$$

Consider the result obtained for the transition t_1 of our example:

$$t_1' = \mathsf{D}_1 \xrightarrow{e_1 \ [\gamma]/\textsc{Exit}(\mathit{root}') \hat{} \ \langle e_{t_1} \rangle \hat{} \ \textsc{Entry-Or}(\mathit{root}',\, \{\mathsf{C}_{11}\})} I_{\mathsf{and}} \quad , \qquad (4.26)$$

where γ has been shown in (4.18).

Trans' is the set of all action transitions, history transitions and scheduling transitions. Assuming that guards are not translated to DNF formulæ there are at most $3|State_{\mathsf{and}}| + |Trans|$ new transitions in the flat model and each of them is at most $2|State_{\mathsf{and}}|$ times long. The size change is within the polynomial bounds in the size of hierarchical model, or more precisely in $O(|State|^2 + |State| \cdot |Trans|)$. The size of the flat statechart is bounded by the square of the size of the original hierarchical statechart. As we argued previously, expansion of guards to DNF clauses and the respective transition multiplication incurres only polynomial size increase for realistic models (and this costs only affects variants of statecharts with restrcited guard syntax as visualSTATE).

4.5.2 Example Results

Fig. 4.15 presents an overview of the flat statechart produced by applying the algorithm of section 4.5.1 to the model of Fig. 2.1. The notation is imprecise, aiming at intuitive explanation. More precise account of the transformation can be found on Fig. 4.16.

4.5.3 Code Generation

Although polynomial, the algorithm of the previous section remains rather inefficient. We would like to eliminate excess transitions, simplify guards, and reduce the number of administrative signals. The resulting flat rulesets

External and internal events:

$Event' = \{e_1, e_2\}$

$Signal' = \{s_1, e_A^{en}, e_B^{en}, e_{C_1}^{en}, e_{C_2}^{en}, e_{D_1}^{en}, e_{D_2}^{en}, e_{E_1}^{en}, e_{C_{11}}^{en}, e_{C_{12}}^{en}, e_{D_{11}}^{en}, e_{D_{12}}^{en}, e_{E_{11}}^{en}, e_{E_{12}}^{en},$

$\qquad\quad e_A^{ex}, e_B^{ex}, e_{C_1}^{ex}, e_{C_2}^{ex}, e_{D_1}^{ex}, e_{D_2}^{ex}, e_{E_1}^{ex}, e_{C_{11}}^{ex}, e_{C_{12}}^{ex}, e_{D_{11}}^{ex}, e_{D_{12}}^{ex}, e_{E_{11}}^{ex}, e_{E_{12}}^{ex}, e_D^h,$

$\qquad\quad e_{t_1}, e_{t_2}, e_{t_3}\}$

Exit transitions:

$\mathsf{I_{and}} \xrightarrow{e_A^{ex}\ [A]/ex(A)} \mathsf{I_{and}}, \mathsf{I_{and}} \xrightarrow{e_B^{ex}\ [B]/ex(B)} \mathsf{I_{and}}, \mathsf{I_{and}} \xrightarrow{e_{C_1}^{ex}\ [C_1 \wedge A]/ex(C_1)} \mathsf{I_{and}},$

$\mathsf{I_{and}} \xrightarrow{e_{C_2}^{ex}\ [C_2 \wedge A]/ex(C_2)} \mathsf{I_{and}}, \mathsf{I_{and}} \xrightarrow{e_{D_1}^{ex}\ [D_1 \wedge B]/ex(D_1)} \mathsf{I_{and}}, \mathsf{I_{and}} \xrightarrow{e_{D_2}^{ex}\ [D_2 \wedge B]/ex(D_2)} \mathsf{I_{and}},$

$\mathsf{I_{and}} \xrightarrow{e_{E_1}^{ex}\ [E_1 \wedge B]/ex(E_1)} \mathsf{I_{and}}, \mathsf{I_{and}} \xrightarrow{e_{C_{11}}^{ex}\ [C_{11} \wedge C_1 \wedge A]/ex(C_{11})} \mathsf{I_{and}},$

$\mathsf{I_{and}} \xrightarrow{e_{C_{12}}^{ex}\ [C_{12} \wedge C_1 \wedge A]/ex(C_{12})} \mathsf{I_{and}}, \mathsf{I_{and}} \xrightarrow{e_{D_{11}}^{ex}\ [D_{11} \wedge D_1 \wedge B]/ex(D_{11})} \mathsf{I_{and}},$

$\mathsf{I_{and}} \xrightarrow{e_{D_{12}}^{ex}\ [D_{12} \wedge D_1 \wedge B]/ex(D_{12})} \mathsf{I_{and}}, \mathsf{I_{and}} \xrightarrow{e_{E_{11}}^{ex}\ [E_{11} \wedge E_1 \wedge B]/ex(E_{11})} \mathsf{I_{and}},$

$\mathsf{I_{and}} \xrightarrow{e_{E_{12}}^{ex}\ [E_{12} \wedge E_1 \wedge B]/ex(E_{12})} \mathsf{I_{and}}$

Entry transitions:

$\mathsf{I_{and}} \xrightarrow{e_A^{en}\ [true]/en(A)} \mathsf{A}, \mathsf{I_{and}} \xrightarrow{e_B^{en}\ [true]/en(B)} \mathsf{B}, \mathsf{I_{and}} \xrightarrow{e_{C_1}^{en}\ [A]/en(C_1)} \mathsf{C_1},$

$\mathsf{I_{and}} \xrightarrow{e_{C_2}^{en}\ [A]/en(C_2)} \mathsf{C_2}, \mathsf{I_{and}} \xrightarrow{e_{D_1}^{en}\ [B]/en(D_1)} \mathsf{D_1}, \mathsf{I_{and}} \xrightarrow{e_{D_2}^{en}\ [B]/en(D_2)} \mathsf{D_2},$

$\mathsf{I_{and}} \xrightarrow{e_{E_1}^{en}\ [B]/en(E_1)} \mathsf{E_1}, \mathsf{I_{and}} \xrightarrow{e_{C_{11}}^{en}\ [C_1 \wedge A]/en(C_{11})} \mathsf{C_{11}},$

$\mathsf{I_{and}} \xrightarrow{e_{C_{12}}^{en}\ [C_1 \wedge A]/en(C_{12})} \mathsf{C_{12}}, \mathsf{I_{and}} \xrightarrow{e_{D_{11}}^{en}\ [D_1 \wedge B]/en(D_{11})} \mathsf{D_{11}},$

$\mathsf{I_{and}} \xrightarrow{e_{D_{12}}^{en}\ [D_1 \wedge B]/en(D_{12})} \mathsf{D_{12}}, \mathsf{I_{and}} \xrightarrow{e_{E_{11}}^{en}\ [E_1 \wedge B]/en(E_{11})} \mathsf{E_{11}}, \mathsf{I_{and}} \xrightarrow{e_{E_{12}}^{en}\ [E_1 \wedge B]/en(E_{12})} \mathsf{E_{12}}$

Action transitions (for original hierarchical transitions):

$\mathsf{I_{and}} \xrightarrow{e_{t_1}\ [true]/\langle o_1, s_1 \rangle} \mathsf{I_{and}}, \mathsf{I_{and}} \xrightarrow{e_{t_2}\ [true]/\langle o_2 \rangle} \mathsf{I_{and}},$

$\mathsf{I_{and}} \xrightarrow{e_{t_3}\ [true]/\langle \rangle} \mathsf{I_{and}}, \mathsf{I_{and}} \xrightarrow{e_{t_4}\ [true]/x = (x + n)\%2} \mathsf{I_{and}}$

History entries:

$\mathsf{I_{and}} \xrightarrow{e_D^h\ [D_1]/\langle e_{D_1}^{en}, e_{D_{11}}^{en} \rangle} \mathsf{I_{and}}, \mathsf{I_{and}} \xrightarrow{e_D^h\ [D_2]/\langle e_{D_2}^{en} \rangle} \mathsf{I_{and}}$

Schedule transitions (bold on the diagram 4.15):

$t_1' = \mathsf{D_1} \xrightarrow{e_1\ [(D_1 \wedge B) \wedge ((D_{12} \wedge D_1 \wedge B) \wedge \neg(E_{11} \wedge E_1 \wedge B))]/}$

$\qquad\qquad \xrightarrow{\langle e_{C_{11}}^{ex}, e_{C_{12}}^{ex}, e_{C_1}^{ex}, e_{C_2}^{ex}, e_A^{ex}, e_{D_{11}}^{ex}, e_{D_{12}}^{ex}, e_{D_1}^{ex}, e_{D_2}^{ex}, e_{E_{11}}^{ex}, e_{E_{12}}^{ex}, e_{E_1}^{ex}, e_B^{ex}, e_{t_1}, e_A^{en}, e_{C_1}^{en}, e_{C_{11}}^{en} \rangle} \mathsf{I_{and}}$

$t_2' = \mathsf{A} \xrightarrow{s_1\ [A \wedge \neg(C_2 \wedge A)]/}$

$\qquad\qquad \xrightarrow{\langle e_{C_{11}}^{ex}, e_{C_{12}}^{ex}, e_{C_1}^{ex}, e_{C_2}^{ex}, e_A^{ex}, e_{D_{11}}^{ex}, e_{D_{12}}^{ex}, e_{D_1}^{ex}, e_{D_2}^{ex}, e_{E_{11}}^{ex}, e_{E_{12}}^{ex}, e_{E_1}^{ex}, e_B^{ex}, e_{t_2}, e_b^{en}, e_D^h, e_{D_{11}}^{en}, e_{E_1}^{en}, e_{E_{11}}^{en} \rangle} \mathsf{I_{and}}$

$t_3' = \mathsf{C_2} \xrightarrow{e_2\ [(C_2 \wedge A)\ (x == 1)]/}$

$\qquad\qquad \xrightarrow{\langle e_{C_{11}}^{ex}, e_{C_{12}}^{ex}, e_{C_1}^{ex}, e_{C_2}^{ex}, e_A^{ex}, e_{D_{11}}^{ex}, e_{D_{12}}^{ex}, e_{D_1}^{ex}, e_{D_2}^{ex}, e_{E_{11}}^{ex}, e_{E_{12}}^{ex}, e_{E_1}^{ex}, e_B^{ex}, e_{t_3}, e_B^{en}, e_{D_1}^{en}, e_{D_{12}}^{en}, e_{E_1}^{en}, e_{E_{11}}^{en} \rangle} \mathsf{I_{and}}$

$t_4' = \mathsf{C_1} \xrightarrow{e_2\ [C_1 \wedge A]/\langle e_{C_{11}}^{ex}, e_{C_{12}}^{ex}, e_{C_1}^{ex}, e_{t_4}, e_{C_2}^{en} \rangle} \mathsf{I_{and}}$

Figure 4.16: The complete ruleset produced during flattening. Observe the inefficiencies, which can be removed using techniques of section 4.5.3, especially redundant checks in guards and use of a long signal queue.

are extremely simple to represent and interpret using a minimal runtime system.

We will use ϕ be a formula overapproximating the set of reachable configurations of hierarchical model, as seen before.

Administrative signals

Massive use of signals is a disadvantage at runtime as it demands writable memory for maintenance of the signal queue. We get rid of administrative signals by exploiting the fact that code generation is actually solving a simpler problem than the generic flattening defined above. The main difference is that at runtime transitions are processed and fired in some fixed deterministic order. This order may substitute a signal queue in guaranteeing proper schedules of action transitions. For instance the t_1 transition of Fig. 2.1 can be flattened to:

$$
\text{exit:}\quad
\begin{cases}
\mathsf{D_{11}} \xrightarrow{e_1 \ [\mathsf{D_{11}} \wedge \mathsf{D_1} \wedge (\mathsf{D_{12}} \wedge \neg \mathsf{E_{11}})]/ex(\mathsf{D_{11}})} \emptyset \\[2pt]
\mathsf{D_{12}} \xrightarrow{e_1 \ [\mathsf{D_{12}} \wedge \mathsf{D_1} \wedge (\mathsf{D_{12}} \wedge \neg \mathsf{E_{11}})]/ex(\mathsf{D_{12}})} \emptyset \\[2pt]
\mathsf{D_2} \xrightarrow{e_1 \ [\mathsf{D_2} \wedge \mathsf{D_1} \wedge (\mathsf{D_{12}} \wedge \neg \mathsf{E_{11}})]/ex(\mathsf{D_2})} \emptyset \\[2pt]
\mathsf{D_1} \xrightarrow{e_1 \ [\mathsf{D_1} \wedge (\mathsf{D_{12}} \wedge \neg \mathsf{E_{11}})]/ex(\mathsf{D_1})} \emptyset \\[2pt]
\mathsf{E_{11}} \xrightarrow{e_1 \ [\mathsf{E_{11}} \wedge \mathsf{D_1} \wedge (\mathsf{D_{12}} \wedge \neg \mathsf{E_{11}})]/ex(\mathsf{E_{11}})} \emptyset \\[2pt]
\mathsf{E_{12}} \xrightarrow{e_1 \ [\mathsf{E_{12}} \wedge \mathsf{D_1} \wedge (\mathsf{D_{12}} \wedge \neg \mathsf{E_{11}})]/ex(\mathsf{E_{12}})} \emptyset \\[2pt]
\mathsf{E_1} \xrightarrow{e_1 \ [\mathsf{E_1} \wedge \mathsf{D_1} \wedge (\mathsf{D_{12}} \wedge \neg \mathsf{E_{11}})]/ex(\mathsf{E_1})} \emptyset \\[2pt]
\mathsf{B} \xrightarrow{e_1 \ [\mathsf{B} \wedge \mathsf{D_1} \wedge (\mathsf{D_{12}} \wedge \neg \mathsf{E_{11}})]/ex(\mathsf{B})} \emptyset
\end{cases}
\tag{4.27}
$$

$$
\text{actions:}\quad
\left\{ \mathsf{D_1} \xrightarrow{e_1 \ [\mathsf{D_1} \wedge (\mathsf{D_{12}} \wedge \neg \mathsf{E_{11}})]/\langle o_1, s_1 \rangle} \emptyset \right.
\tag{4.28}
$$

$$
\text{entry:}\quad
\left\{ \mathsf{D_1} \xrightarrow{e_1 \ [\mathsf{D_1} \wedge (\mathsf{D_{12}} \wedge \neg \mathsf{E_{11}})]/en(\mathsf{A})\,\hat{}\,en(\mathsf{C_1})\,\hat{}\,en(\mathsf{C_{11}})} \emptyset \right.
\tag{4.29}
$$

According to the semantic rules of chapter 2, the empty set of targets of the flat transition means that the active state configuration remains unchanged. Although similar to loop transitions, the targetless transitions are more efficient to execute.

The above sequence has been obtained by instantiating a transition for every signal of the schedules of t_1'. Its top–down interpretation corresponds to firing t_1'. In the generic algorithms of the previous section we would use a signal queue to guarantee this sequencing. Presently no administrative signals are used, but guards are evaluated multiple times and entry/exit transitions cannot be reused.

Guard Analysis

Guard analysis may be used to eliminate superfluous transitions and computations. Note that the *flat* function produces guards with redundant con-

ditions (see γ in (4.18)). Such guards can be improved, by optimization performed under the assumption that formula ϕ holds. This automatically removes references to constantly true variables, superfluous states (such us E_1, which is equivalent to E) and repeating clauses. Removal of variables should be accompanied by removal of corresponding states from the flat structure. There is no need to represent them at runtime. For example states E_1 and $\mathsf{I}_{\mathsf{and}}$ (and their parents) would be eliminated and guards simplified respectively.

While performing guard analysis, unsatisfiable guards can be found on transitions. Transitions containing them can be discarded. For example the exit transition from state D_2 in (4.27) will never fire as D_1 and D_2 may never be active at the same time according to ϕ. SCOPE relies on a BDD engine in the implementation of guard analysis.

Merging Transitions

Some transitions in the sequence can be merged saving space and increasing the execution speed. Whenever guards of two subsequent transitions are equivalent (under ϕ), the two rules can be merged into a single one whose action is the concatenation of original actions and whose set of targets is the union of the original sets. A special case, which is particularly easy to detect, is that of the last two entries in every group implementing a single transition. For example transitions (4.28) and (4.29) can be combined yielding:

$$\mathsf{D}_1 \xrightarrow{\ e_1 \ [\mathsf{D}_1 \wedge (\mathsf{D}_{12} \wedge \neg \mathsf{E}_{11})]/\langle o_1 \rangle \char`^ en(\mathsf{A}) \char`^ en(\mathsf{C}_1) \char`^ en(\mathsf{C}_{11}) \char`^ \langle s_1 \rangle\ } \emptyset \qquad (4.30)$$

Although much more complex, merging is also applied to transitions realizing exit schedules. Exit transitions, or parts of thereof, can be joined if there is some firm knowledge about the source states of a given transition (inferred from the guard). In the extreme case, when the knowledge about the source and target states is complete the transition gets translated to a single rule. This is for example the case with t_3:

$$\mathsf{C}_2 \xrightarrow{\ e_2 \ [\mathsf{C}_2 \wedge \mathsf{C}_1 \wedge \mathsf{A}]/ex(\mathsf{C}_2) \char`^ ex(\mathsf{C}_1) \char`^ ex(\mathsf{A}) \char`^ en(\mathsf{B}) \char`^ en(\mathsf{D}_1) \char`^ en(\mathsf{D}_{12}) \char`^ en(\mathsf{E}_1) \char`^ en(\mathsf{E}_{11})\ } \emptyset \quad . \qquad (4.31)$$

4.5.4 Evaluation

The algorithm has been evaluated using both artificial and real industrial examples. Efficiency has been measured against the hierarchical back-end of SCOPE, which in turn performs slightly better than the implementation of IAR visualSTATE. Executable sizes and execution times are reported for skeleton control programs with dummy action and guard functions, compiled by gcc ver. 3.2 targeting x86 PC (see Tables 4.3–4.4). Execution time has been measured for feeding the compiled system with 10^7 random events in
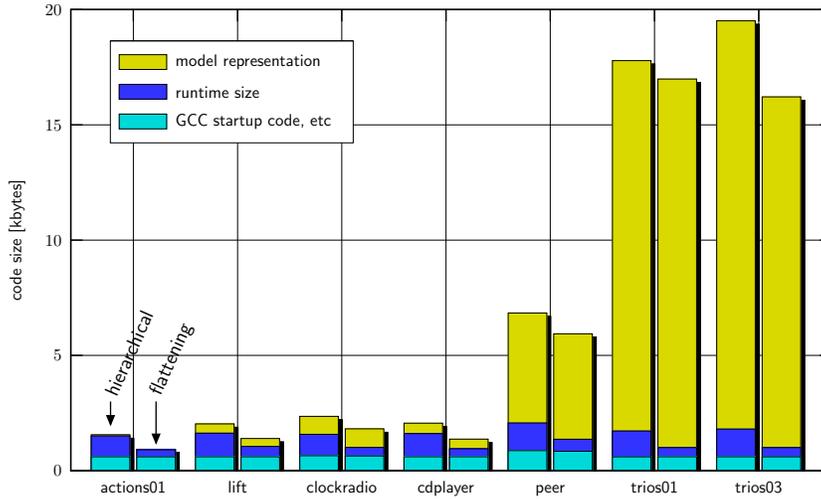
Model	states	trans.	depth	executable size		
				FL-CG	HI-CG	ratio
actions01	4	1	3	3 036	3 704	0.82
lift	18	19	3	3 644	4 372	0.83
clockradio	20	27	7	4 108	4 652	0.88
cdplayer	21	16	7	3 560	4 312	0.83
peer	275	192	23	9 252	10 536	0.88
trios01	1121	840	9	20 772	24 108	0.86
trios03	1121	840	9	19 972	24 684	0.81

Table 4.3: Size results: hierarchical code generation vs flattening-based code generation. FL-CG denotes code generation for flat models, HI-CG denotes direct code generation for hierarchical models. Executable sizes in bytes
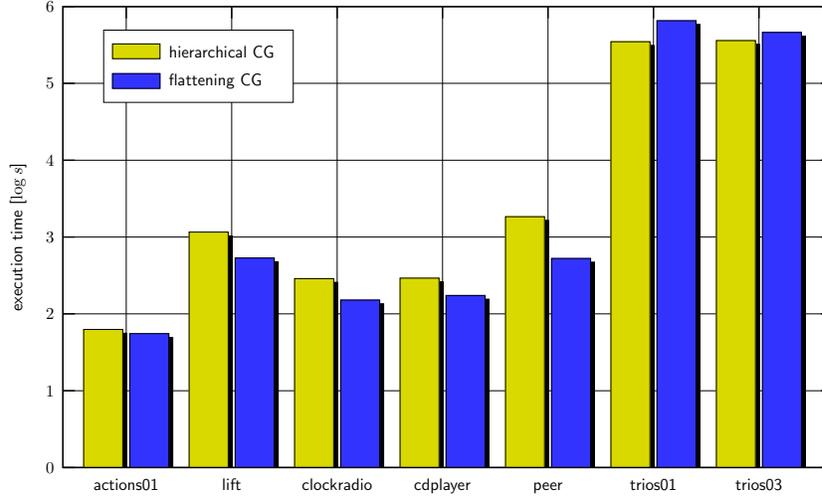
Model	states	trans.	depth	execution time		
				FL-CG	HI-CG	ratio
actions01	4	1	3	5.71	6.03	0.95
lift	18	19	3	15.29	21.41	0.71
clockradio	20	27	7	8.84	11.69	0.76
cdplayer	21	16	7	9.38	11.77	0.80
peer	275	192	23	15.19	26.16	0.58
trios01	1121	840	9	335	255	1.31
trios03	1121	840	9	288	259	1.10

Table 4.4: Speed results: hierarchical code generation vs flattening-based code generation. FL-CG denotes code generation for flat models, HI-CG denotes direct code generation for hierarchical models. Running times of the generated code in seconds.

Model	H8 executable size		
	FL-CG	HI-CG	ratio
actions01	1 312	1 906	0.69
lift	1 750	2 356	0.74
clockradio	2 116	2 640	0.80
cdplayer	1 700	2 344	0.73
peer	6 298	7 226	0.87
trios01	17 670	18 326	0.96
trios03	16 878	20 064	0.84

Table 4.5: Sizes of executables cross-compiled with h8300-hms-gcc 3.3.2.

	AVR executable size			AVR runtime vs model size			
Model	FL-CG	HI-CG	ratio	FL rtime	FL model	HI rtime	HI model
actions01	952	1 592	0.60	313	25	925	50
lift	1 428	2 076	0.69	459	353	1 047	412
clockradio	1 856	2 412	0.77	377	826	947	800
cdplayer	1 400	2 108	0.66	353	431	1 027	465
peer	6 076	7 004	0.87	530	4 684	1 226	4 888
trios01	17 396	18 212	0.96	409	16 371	1 143	16 454
trios03	16 604	19 980	0.83	409	15 580	1 229	18 134

Table 4.6: Sizes of the runtime interpreter and the runtime model representation for gcc-avr compiled code. These numbers are included in the complete executables of the left side. Approximately 600 bytes is left out in each line, used by gcc for internal initialization code.

repetitive series (run on Linux, 450MHz Pentium II). More experiments has been carried out with GCC for AVR and H8/300, exhibiting similar results (see tables 4.5–4.6). This confirms the expectation that the code generated from statecharts is hard for compilers to optimize and there is a need for advanced analysis and transformations on the model level.

The first model, *actions01*, is the smallest model which can be built with the visualSTATE design environmnet. It contains two states and a single transition. This model exhibits the difference between sizes of the two runtime libraries. The difference does not increase for bigger examples—both algorithms scale well. However, the flattened code is faster, simpler and smaller, despite the fact that less engineering effort had been put in the implementation of the specific parts of the runtime. No size explosions are visible for big models (*trios01*, *trios03*). *Peer*, the biggest real life example, a complicated model of a very advanced coffee vending machine exhibits a particularly good result, which is perhaps the best recommendation of our flattening.

The *lift* example is a flat statechart and as such is not affected by flattening. Still the flattened version operates much faster than hierarchical one. This is because the hierarchical interpreter is much more sophisticated than the flat one, and lots of this sophistication is useless for this model. Flat models constitute an important class of models, appreciated by engineers for its simplicity and good applicability to small size tasks. It is thus comforting that a single code generation scheme, the one based on flattening, performs well for both flat and hierarchical models.

	[bytes]
current event (global)	3
state representation (global)	8
stack	30
model variables (global)	4
TOTAL [bytes]	41+4

Table 4.7: RAM usage in SCOPE: quick generous estimate for a simple thermostat model, assuming 8bit word, 32bit addressing. The stack usage is nearly half smaller with more common 16-bit addressing

The last two models (from *trios* series) are artificial examples. They consist of triples of and-states and or-states interleaved several times. They resemble our (α, β)-models of section 4.4.1, which have been used to demonstrate the superpolynomial nature of flattening in absence of sequential message passing. As expected, no explosion is observed in the present results, where we exploit the sequential nature of code generation. Such highly concurrent models are hardly met in industrial applications, which makes the slow-down reported not essential. Also none of the industrial cases we know suffers from to slow interpretation of the generated control code.

Last but not the last, SCOPE is very conservative about using writable memory at runtime. Table 4.7 shows a summary of RAM usage for a simple thermostat model. Note that only the third entry denotes the memory used by SCOPE for internal purposes.

4.5.5   Correctness Sketch

We shall briefly sketch the correctness argument for the flattening algorithm presented above. We require that the priority ordering on states ◄ of the hierarchical statechart (determining the traversal orders) is total and fixed, and that the entry schedules of all flat transitions have been generated in agreement with this ordering. The priority ordering on transitions ◁ is assumed to be entirely nondeterministic (empty) for both flat and hierarchical statecharts (in practical implementations though, both would be fixed and in agreement).

We will assume that $\mathcal{S}'$ is a flat statechart that has been obtained from the hierarchical statechart $\mathcal{S}$ by applying our flattening algorithm, agreeing with ◄ ordering. We begin with defining an auxiliary correspondence relation for well-formed global states of hierarchical and flat statecharts:

Definition 4.7. *Let* $\mathcal{S}' = (\Gamma_E, \Gamma_F, \Gamma_V, Signal', Var_E, Var_I, State', \searrow', ini', his', ex',$ *$en', Trans')$ be a flat statechart and* $\mathcal{S} = (\Gamma_E, \Gamma_F, \Gamma_V, Signal, Var_E, Var_I, State, \searrow,$

ini, his, ex, en, Trans) be a hierarchical statechart such that:

1. $State'_{or} = State_{or} \cup \{I_{or}, root_{or}\}$
2. $State'_{and} = State_{and} \cup \{I_{and}, root_{and}\}$
3. $\forall s_1 \in State_{or}.\forall s_2 \in State_{and}.\ s_1 \searrow s_2 \Rightarrow s_1 \searrow' s_2$
4. $\forall s \in State_{or}.\ root_{and} \searrow' s_1$
5. $root_{or} \searrow' root_{and} \wedge I_{or} \searrow' I_{and} \wedge root_{and} \searrow' I_{or}$
6. $dom(his') = \emptyset \wedge dom(ex') = \emptyset \wedge dom(en') = \emptyset$

The pair of state components (σ', ϱ') *of* $\mathcal{S}'$ *corresponds to the triple* (σ, ϱ, his) *of state components of a hierarchical statechart , written* $(\sigma', \varrho') \doteq (\sigma, \varrho, his)$, *iff*

7. $\sigma \subseteq \sigma' \wedge \varrho' = \varrho \wedge rng(his) \subseteq \sigma'$
8. $\forall s \in \sigma'.s \neq I_{and} \Rightarrow (s \in \sigma \vee descend^+(s) \cap \sigma \neq \emptyset \vee ancest^+(s) \cap State_{and} \nsubseteq \sigma')$,

where descend and ancest are defined using $\searrow$ *(and not* $\searrow'$ *).*

Proposition 4.8. *If* $(\sigma', \varrho', his') \doteq (\sigma, \varrho, his)$ *then* $\forall g \in Guard.\ (\sigma \models g \iff \sigma' \models flat(g))$ *and* $\forall c \in Exp.\ (\varrho \models exp \iff \varrho' \models exp)$.

Proposition 4.9. *If transition* t' *of* $\mathcal{S}'$ *is a flattened version of transition* t *of* $\mathcal{S}$ *(a schedule transition) and* $(\sigma', \varrho', his') \doteq (\sigma, \varrho, his)$ *and signal queues of both models have identical content* $q' = q$ *then* $t' \in enabled(\mathbf{hd}\ q', \sigma', \varrho') \iff t \in enabled(\mathbf{hd}\ q, \sigma, \varrho)$.

Observe that firing a single transition in the hierarchical model executes a sequence of actions (ignore the possibility of triggering local signals for now). Firing a flattened transition does not execute any actions in the first step, but it only places administrative signals in the signal queue. The second run in the very same macrostep over these signals actually executes actions. So if there are no signals on states and transitions then the outputs produced by both models are the same. The following proposition generalizes these observation for all enabled transitions:

Proposition 4.10. *If* $(\sigma'_0, \varrho'_0, his'_0) \doteq (\sigma_0, \varrho_0, his_0)$ *and the* $\mathcal{S}$ *does not contain any signals by itself, then:*

$$\text{whenever } \langle \sigma'_0, \varrho'_0, his_0 \rangle \xrightarrow{e(v_1,...,v_m)\ \ os'!} \langle \sigma'_1, \varrho'_1, his'_1 \rangle$$

$$\text{then also } \langle \sigma_0, \varrho_0, his_0 \rangle \xrightarrow{e(v_1,...,v_m)\ \ os!} \langle \sigma_1, \varrho_1, his_1 \rangle$$

$$\text{and } (\sigma'_1, \varrho'_1, his'_1) \doteq (\sigma_1, \varrho_1, his_1) \ .$$

All transitions are translated to a long schedule of signals in the first step in some nondeterministic order, which can map the original order. Then the schedule is discharged producing exit, entry and transition actions in agreement with ◄, leading to new state which is in correspondence with the new hierarchical state.

After the warm-up we shall now allow signals in the source model and consider the execution of macrosteps in the flat model. All events and signals of the original model (hierarchical microsteps) are now always processed in two flat microsteps. As an example consider a system where two transitions fire in reaction to an event *e* and
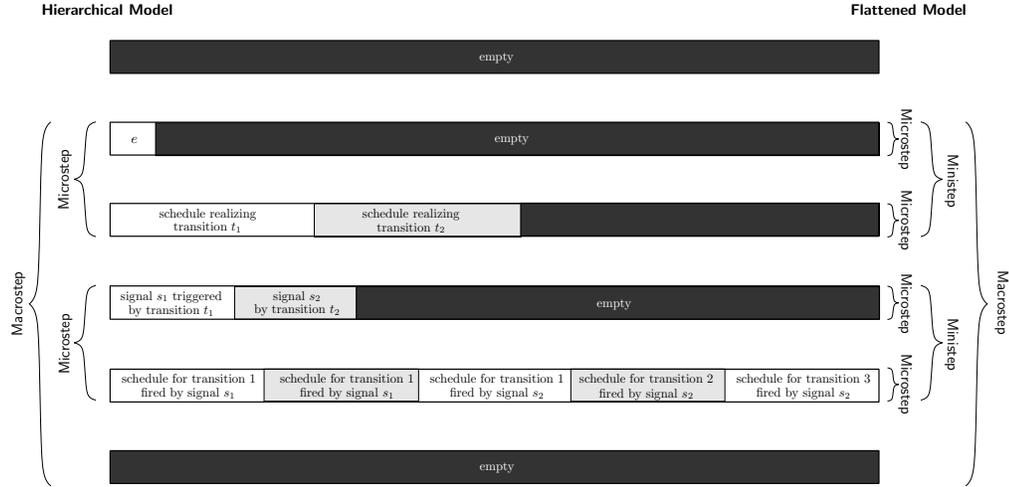
Figure 4.17: Ministeps of the flat statecharts. Each bar represents the state of the signal queue after a single microstep of the flat model. Each hierarchical microstep takes two flat microsteps.

then each of them triggers a single signal that has to be processed in the subsequent microstep. The reactions to these signals only produce outputs, no more signals. Figure 4.17 demonstrates the content of signal queue after each microstep in the flat model. Two consecutive microsteps constitute a ministep: produce a schedule and discharge it. We can prove that ministeps preserve correspondence relation $\doteq$.

Definition 4.11. *A* ministep *is a sequence of microsteps of a flat statechart iterated until the queue only contains elements of Signal (no administrative signals).*

Proposition 4.12. *If two global states of $\mathcal{S}$ and $\mathcal{S}'$ correspond and both models have the same signal queue then the microstep of $\mathcal{S}$ produces the same outputs as the ministep of $\mathcal{S}'$ and the same signal queue.*

The above proposition directly generalizes to the macrostep level.   □

A similar theorem can also be proved for bisimulation, if both priority orderings on states and transitions ◄ and ◁ are total since in such case the models are deterministic.

4.6   Related Work

Most of the material presented in this chapter has been previously published. The hierarchical code generator was described in [130], the lower bound on flattening was proved in [133, 134]. The efficient polynomial flattening algorithm and its practical implementation were presented in [132]. It is the first time though, that this material has been put together in a uniform framework, using a fairly complete language of statecharts.

There is a multitude of research-based statechart translators available. Most of them take hierarchical code generation approach, without performing more significant optimizations [138, 116, 65]. Much less care is taken to make the implementation efficient. The focus is more on code readability an use of natural constructs of target language than efficiency. The usefulness of such tools for constrained embedded systems is not usually considered.

Erpenbach [32] in his dissertation focuses mostly on the worst case reaction time analysis, proposing only a very simple hierarchical representation based on switch statements. He addresses the cost of double-buffering problem for variables, proposing an optimization which reorders the transitions to decrease the need for buffering—so that assignments happen after read accesses, if possible. Our flattening algorithm is fully compatible with his approach, because we have only demanded that rules within a group implementing a single hierarchical transition maintain a specific order. We do not impose any restrictions on the relative order among the groups themselves, which suffices for Erpenbach's algorithm. In other words Erpenbach only requires control over the $\lhd$ ordering, which we do not constrain.

Björklund, Lilius and Porres [13] devise an intermediate language, that should be compilable efficiently. Nevertheless, the use of flattening in the course of translation indicates possible exponential growth of code (they do not give any evidence that their flattening is polynomial, nor what equivalence or refinement their transformation preserves).

The hierarchical code generator was mostly inspired by implementation of Behrmann and others [7]. I have abandoned one of their main ideas though: memoization of guard values (to speed up condition evaluation). I recognize it unsuitable for constrained systems, where writable memory is a scarce resource. Instead I provided runtime representation, which can recompute guards at low cost.

Drusinsky [31] and Ramesh [110] discuss the state encoding problem from hardware implementation perspective. Both encodings proposed are more compact than those presented here. A hardware implementation can very efficiently extract single bits and groups of bits from the state register, whereas this seems to be expensive in software. Apart from this difference Drusinsky's encoding is very much alike to the flag-based encoding of the hierarchical back-end in SCOPE. Both papers are very brief on explaining the structure of combinational block which implements the reaction relation. The applicability of these results for software synthesis should still be investigated.

One such related investigation was undertaken by Jacobsen in his already mentioned thesis [60]. Jacobsen represented statecharts (both current configuration and the transition relation) in Binary Decision Diagrams, which is really similar to implementing them in a combinational block. He used a BDD package and a simple loop as a runtime engine. Despite trying various variable ordering his conclusions were disappointing. Both the representa-

tion and the runtime engine were significantly bigger from the kernel and
rule tables of visualSTATE, with which SCOPE competes well both in the
hierarchical and flattening version.

My CONFIGURATION-BOUND algorithm approximating cardinality of the
active basic states set can be seen as a simpler and weaker version of Drusin-
sky's algorithm for finding maximum-cardinality exclusivity set presented
in [31].

Both the existence of polynomial flattening algorithm and the efficiency
of the implementation are somewhat surprising as numerous authors pre-
sented flattening algorithms suffering from size explosion or informally con-
jectured about the superpolynomial hardness of this problem [130, 26, 15,
5, 16, 112].

The impact of introducing hierarchy in statecharts has been studied
previously [3, 5], however only questions relevant to the model checking
community have been addressed. Our developments discuss succinctness of
hierarchical models from the program synthesis perspective.

Alur et al. [3] thoroughly discuss the impact of hierarchy on model check-
ing problems and the size of models. Sadly, or fortunately for us, they omit
the relation between concurrent hierarchical models and flat models in our
sense, thus their results cannot be directly used to state the hardness of flat-
tening. Moreover Alur and colleagues exploit the sharing of subhierarchies
in their semantics, which is not commonly used in engineering modeling
languages (see UML statecharts).

David et al. [26] claim that flattening a hierarchical transition with their
algorithm may lead to an exponential growth of the model in the depth
of the structure. Note that it exactly agrees with formula 4.9 presented
above. Thus their algorithm can be used as another argument explaining
observation 4.5. The question of establishing strict bounds for arbitrary
models in the size of the model still remains open.

Drusinsky and Harel [30] discuss the succinctness introduced by cooper-
ative concurrency, however they do not consider the influence of hierarchy
on succinctness. The present result explores a different dimension of their
succinctness space for statecharts.

4.7   Beyond the Basics

Our polynomial flattening algorithm relied on the sequential handling of sig-
nals in visualSTATE (and UML). Other variants of statecharts, most notably
[42, 108] and [86], offer alternative signal handling semantics, where all sig-
nals produced in a single microstep are processed simultaneously in the next
microstep. Such semantics drastically reduces the control we have over the
order of processing. We conjecture that an algorithm similar to ours cannot
be used for original Harel's statecharts. Thus the result of theorem 4.4 is

likely to be strengthened, by allowing set-based signal communication in the target language. This remains the main open question in the future work.

Our execution engine for flat statecharts (section 4.3) very much resembles the operation of the most restricted industrial kernels for embedded systems. I had the pleasure of examining such a proprietary kernel used in laboratories of Danfoss A/S in development of control programs for cooling devices. The conclusion was that it would be rather straightforward to port SCOPE's back-end to target this very kernel. The expected sizes of generated programs should be close to those using the flattening back-end of current SCOPE. Danfoss models are restricted in nature. This makes their kernel smaller than our runtime engine. At the same time their models can be represented more compactly, because they are written as low level linker scripts (which we have not used in SCOPE to warrant higher portability). Motivated by the compactness of the generated code, the company expressed interest in developing a port targeting their kernel.

As we have mentioned in section 4.1, current treatment of signals in the model-checker of visualSTATE is very inefficient. Improvements in treatment of communication in this model-checker are under scrutiny in a newly established research project between IAR Systems A/S and CISS, at Aalborg University. Efficient analysis of models with queues will not only boost some proofs of correctness, but will also enable many optimizations in code generation that cannot be otherwise conducted on models.

4.8   Summary

In this chapter we have thoroughly explained the works of two major modes of SCOPE code generator: one based on preserving the hierarchy tree at runtime, and one based on flattening. The hierarchical engine has been shown to perform satisfactorily, which is important because its code generation algorithm is rather direct, so it is easier to trust it. As of my best knowledge this was the first hierarchical code generator explicitly targeting efficiency and conservative usage of resources. Nevertheless the new sophisticated flattening algorithm beats it for virtually all models, while enjoying a very simple execution engine at runtime.

On the theoretical side we have shown the lower bound on flattening in absence of message passing in the target language. This proof, though not very instructive, inspired us to design an efficient flattening algorithm: it hinted that one should use message passing internally. A number of actual flattening algorithms have been indicated, which face the size explosion issue, which has now been shown to be inherent for the problem, not only for the algorithms themselves.

Our lower bound result presents an argument against code generation techniques for statecharts, which are based on flattening in absence of mes-

sage passing, or any other concept able to enforce the order of execution. Such techniques would be tempting otherwise, since lack of signal communication significantly lowers the usage of writable memory, which is a crucial requirement in many engineering applications, especially in the embedded systems domain.

5

Color-blind Semantics for Environments

As we have argued before the reactive synchronous paradigm seems to be predominant in development of embedded software. In the present chapter we consider the problem of modeling execution environments for such systems in general. Our environments are *color-blind*: they may not be able to distinguish some responses of the system. This property can be exploited in optimization of systems, which goes beyond dead code elimination and early compile-time execution, by permitting some mutations in the system.

One immediate theoretical application of color-blindness, presented in section 5.6, is an elegant modeling of various output structures, an alternative to the techniques presented in section 2.2.3. In chapter 6 they will be used in the modeling and development of software product lines.

Our development unfolds as follows. We begin with the introduction of I/O alternating transition systems, which are abstract models for general reactive synchronous systems. In section 5.2 we extend them with the novel notion of color-blindness. Section 5.3 defines composition operators for color-blind environments, while section 5.4 argues that color-blind equivalence can be used in applications typical for a refinement preorder. In section 5.5 the theoretical framework is instantiated for more realistic languages like statecharts. As an immediate example we show how color-blindness can replace parameterization of output structure (section 5.6) and discuss various notions of discrimination (section 5.7). Finally possible extensions (section 5.8) and the related work (section 5.9) are discussed.

The theory presented here is rather central for the entire work. Still one can safely omit sections 5.5.2, 5.6 and 5.8-5.9, as well as all the proofs, without compromising on comprehension in later sections and chapters.

5.1   I/O Alternating Transition Systems

Two core properties of reactive synchronous systems are input-enabledness (see page 32) and synchronicity (see page 17). A reactive synchronous system can react to any input event at any time. Each reaction occurs in infinitely short time, so that the system is always able to observe the arrival of the next event. *I/O-alternating transition systems* can conveniently be used in formalization of reactive input-enabled semantics:

Definition 5.1. *An I/O-alternating transition system of reactive synchronous processes, or IOATS in short, is a tuple*

$$\mathcal{P} = (In, Out, Gen, Obs, \xrightarrow{!}, \xrightarrow{?}, s^0) \ ,$$

where In and Out are sets of inputs and outputs, Gen is a finite set of generator states (generators), Obs is a finite set of observer states (observers), $\xrightarrow{!} \subseteq Gen \times Out \times Obs$ is a generation relation and $\xrightarrow{?} \subseteq Obs \times In \times Gen$ is an observation relation. The initial state s^0 is a generator or an observer.

We have distinguished two transition relations: $\xrightarrow{!}$ is a generation relation, which advances the process from a generator to an observer state, while $\xrightarrow{?}$ is an observation relation advancing the system from an observer to a generator state. This alternation is inherent to the way synchronous systems operate. We shall write $S \xrightarrow{o!} s$, instead of $(S, o, s) \in \xrightarrow{!}$ and $s \xrightarrow{i?} S$ instead of $(s, i, S) \in \xrightarrow{?}$. The (In, Out) pair is typically referred to as a signature of the transition system. Also we adopt the convention that lower case letters are used for observers and capital letters are used for generators, wherever possible.

Our observers are *input-enabled*, so they never block:

$$\forall s \in Obs. \ \forall i \in In. \ \exists S, o, s'. \ s \xrightarrow{i?} S \wedge S \xrightarrow{o!} s' \tag{5.1}$$

A refinement relation states that a certain IOATS implements a subset of functionality of the other.

Definition 5.2. *Consider two I/O alternating transition systems:*

$$\mathcal{S}_1 = (In, Out, Gen_1, Obs_1, \xrightarrow{!}_1, \xrightarrow{?}_1, s_1^0) \ ,$$

$$\mathcal{S}_2 = (In, Out, Gen_2, Obs_2, \xrightarrow{!}_2, \xrightarrow{?}_2, s_2^0) \ .$$

A binary relation $R \in Obs_1 \times Obs_2$ constitutes a simulation on observers of $\mathcal{S}_1$ and $\mathcal{S}_2$ if $(s_1, s_2) \in R$ implies that:

whenever $s_1 \xrightarrow{i?} S_1 \wedge S_1 \xrightarrow{o!} s_1'$

then for some S_2, s_2' also $s_2 \xrightarrow{i?} S_2 \wedge S_2 \xrightarrow{o!} s_2'$ and $(s_1', s_2') \in R$.

Now let R be the largest of such relations ordered by inclusion. We say that an observer s_2 simulates an observer s_1, written $s_1 \leqslant s_2$, if $(s_1, s_2) \in R$. Finally we say that $\mathcal{S}_2$ simulates $\mathcal{S}_1$, written $\mathcal{S}_1 \leqslant \mathcal{S}_2$, iff $s_1^0 \leqslant s_2^0$.

Let $\mathbb{S}_1$ be an endofunction (domain equals codomain) on binary relations on observers of $\mathcal{S}_1$ and $\mathcal{S}_2$, such that

$$\mathbb{S}_1(R) = \{(s_1, s_2) \mid \forall i, S_1, o, s_1'. \exists S_2, s_2'.$$
$$s_1 \xrightarrow{i?} S_1 \wedge S_1 \xrightarrow{o!} s_1' \Rightarrow s_2 \xrightarrow{i?} S_2 \wedge S_2 \xrightarrow{o!} s_2' \wedge (s_1', s_2') \in R\} \quad (5.2)$$

Proposition 5.3. *A binary relation R constitutes a simulation between observers of two IOATSs $\mathcal{S}_1$ and $\mathcal{S}_2$ iff $R \subseteq \mathbb{S}_1(R)$.*

Proposition 5.4. *$\mathbb{S}_1$ is a monotone endofunction on a complete lattice of binary relations ordered by inclusion.*

By Knaster's and Tarski's [123] fixpoint theorem $\mathbb{S}_1$ has the greatest fixpoint and the simulation relation $\leqslant$ of Def. 5.2 equals this greatest fixpoint, which justifies the correctness of our definition.

An equivalence relation states that two IOATSs are identical (up to a certain criterion):

Definition 5.5. *Consider two I/O alternating transition systems:*

$$\mathcal{S}_1 = (In, Out, Gen_1, Obs_1, \xrightarrow{!}_1, \xrightarrow{?}_1, s_1^0) \ ,$$

$$\mathcal{S}_2 = (In, Out, Gen_2, Obs_2, \xrightarrow{!}_2, \xrightarrow{?}_2, s_2^0) \ .$$

A binary relation $R \in Obs_1 \times Obs_2$ constitutes a bisimulation on observers of $\mathcal{S}_1$ and $\mathcal{S}_2$ if $(s_1, s_2) \in R$ implies that:

whenever $s_1 \xrightarrow{i?} S_1 \wedge S_1 \xrightarrow{o!} s_1'$

then for some S_2, s_2' also $s_2 \xrightarrow{i?} S_2 \wedge S_2 \xrightarrow{o!} s_2'$ and $(s_1', s_2') \in R$,

and whenever $s_2 \xrightarrow{i?} S_2 \wedge S_2 \xrightarrow{o!} s_2'$

then for some S_1, s_1' also $s_1 \xrightarrow{i?} S_1 \wedge S_1 \xrightarrow{o!} s_1'$ and $(s_1', s_2') \in R$.

Now let R be the largest of such relations ordered by inclusion. We say that observers s_1, s_2 are equivalent, written $s_1 \sim s_2$, if $(s_1, s_2) \in R$. Finally we say that IOATSs $\mathcal{S}_1, \mathcal{S}_2$ are equivalent, written $\mathcal{S}_1 \sim \mathcal{S}_2$, iff $s_1^0 \sim s_2^0$.

Let $\mathbb{B}_1$ be an endofunction on binary relations on observers of $\mathcal{S}_1$ and $\mathcal{S}_2$, such that

$$\mathbb{B}_1(R) = \{(s_1, s_2) \mid (\forall i, S_1, o, s_1'. \exists S_2, s_2'. s_1 \xrightarrow{i?} S_1 \wedge S_1 \xrightarrow{o!} s_1'$$

$$\Rightarrow s_2 \xrightarrow{i?} S_2 \wedge S_2 \xrightarrow{o!} s_2' \wedge (s_1', s_2') \in R)$$

$$\wedge (\forall i, S_2, o, s_2'. \exists S_1, s_1'. s_2 \xrightarrow{i?} S_2 \wedge S_2 \xrightarrow{o!} s_2'$$

$$\Rightarrow s_1 \xrightarrow{i?} S_1 \wedge S_1 \xrightarrow{o!} s_1' \wedge (s_1', s_2') \in R)\} \qquad (5.3)$$

Proposition 5.6. *A binary relation R constitutes a bisimulation between observers of two IOATSs $\mathcal{S}_1$ and $\mathcal{S}_2$ iff $R \subseteq \mathbb{B}_1(R)$.*

Proposition 5.7. $\mathbb{B}_1$ *is a monotone endofunction on a complete lattice of binary relations ordered by inclusion.*

By Tarski's fixpoint theorem $\mathbb{B}_1$ has the greatest fixpoint and the simulation relation $\leqslant$ of Def. 5.2 equals this greatest fixpoint, which justifies the correctness of our definition.

We shall distinguish the actual systems and the environments, in which they operate. Environments are free in choice of inputs, while systems independently determine the outputs they produce. A system $\mathcal{S} = (In_\mathcal{S}, Out_\mathcal{S}, Gen_\mathcal{S}, Obs_\mathcal{S}, \xrightarrow{!}_\mathcal{S}, \xrightarrow{?}_\mathcal{S}, s_\mathcal{S})$ operates embedded in some environment $\mathcal{E} = (In_\mathcal{E}, Out_\mathcal{E}, Gen_\mathcal{E}, Obs_\mathcal{E}, \xrightarrow{!}_\mathcal{E}, \xrightarrow{?}_\mathcal{E}, s_\mathcal{E})$. Systems always begin execution in an observer state, so $s_\mathcal{S} \in Obs_\mathcal{S}$, while environments always begin execution in a generator state, so $s_\mathcal{E} \in Gen_\mathcal{E}$. A system $\mathcal{S}$ is *compatible* with an environment $\mathcal{E}$ if $In_\mathcal{S} = Out_\mathcal{E}$ and $Out_\mathcal{S} = In_\mathcal{E}$.

Composition of a system and an environment is defined in the usual way, by synchronization on labels. The initial observer of the system is composed with the initial generator of the environment. Due to the compatibility requirement and input-enabledness of observers, the closed system is able to advance for any input that can be generated by the environment.

$$\frac{E \xrightarrow{i!} e \quad s \xrightarrow{i?} S}{(E, s) \to (e, S)} \qquad \frac{e \xrightarrow{o?} E \quad S \xrightarrow{o!} s}{(e, S) \to (E, s)} \qquad (5.4)$$

For a closed system it is known, which of its states can be exercised by the environment. A given environment may not be able to distinguish two systems from each other, even though they are not identical. We capture this using notions of *relativized simulation* and *relativized bisimulation* between two systems embedded in the same context:

Definition 5.8. *Consider three IOATSs: an environment $\mathcal{E} = (Out, In, Gen, Obs, \xrightarrow{!}, \xrightarrow{?}, E^0)$ and two systems: $\mathcal{S}_1 = (In, Out, Gen_1, Obs_1, \xrightarrow{!}_1, \xrightarrow{?}_1, s_1^0)$ and*

$\mathcal{S}_2 = (In, Out, Gen_2, Obs_2, \overset{!}{\rightarrow}_2, \overset{?}{\rightarrow}_2, s_2^0)$. *A Gen-indexed family of binary relations* $R : Gen \rightarrow \mathcal{P}(Obs_1 \times Obs_2)$ *is a relativized simulation if* $(s_1, s_2) \in R_E$ *implies that:*

whenever $E \overset{i!}{\rightarrow} e \wedge e \overset{o?}{\rightarrow} E'$

then whenever $s_1 \overset{i?}{\rightarrow} S_1 \wedge S_1 \overset{o!}{\rightarrow} s_1'$

then for some S_2, s_2' *also* $s_2 \overset{i?}{\rightarrow} S_2 \wedge S_2 \overset{o!}{\rightarrow} s_2' \wedge (s_1', s_2') \in R_{E'}.$ (5.5)

Let R *be the largest of such families of relations ordered by component-wise inclusion. We say that an observer* s_2 *simulates an observer* s_1 *in the generator* E, *written* $s_1 \leqslant_E s_2$, *if* $(s_1, s_2) \in R_E$. *The system* $\mathcal{S}_2$ *simulates* $\mathcal{S}_1$ *in the context of* $\mathcal{E}$, *written* $\mathcal{S}_1 \leqslant_{\mathcal{E}} \mathcal{S}_2$, *if* $s_1^0 \leqslant_{E^0} s_2^0$.

Let $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{E}$ be IOATSs defined as above. Let $\mathbb{S}_2$ be an endofunction on *Gen*-indexed families of binary relations $Gen \rightarrow \mathcal{P}(Obs_1 \times Obs_2)$ such that $\mathbb{S}_2(R'_E)$ is equal to the set of observer pairs satisfying property (5.5):

$$\mathbb{S}_2(R) = \lambda E. \; \{(s_1, s_2) \mid \forall i, o, e, E', S_1, s_1'. \exists S_2, s_2'. \; s_1 \overset{i?}{\rightarrow} S_1 \wedge S_1 \overset{o!}{\rightarrow} s_1'$$

$$\Rightarrow s_2 \overset{i?}{\rightarrow} S_2 \wedge S_2 \overset{o!}{\rightarrow} s_2' \wedge (s_1', s_2') \in R_{E'}\}$$ (5.6)

Proposition 5.9. *A Gen-indexed family of binary relations* R *consitutes a relativized simulation iff* $R \subseteq \mathbb{S}_2(R)$ *(where inclusion is defined component-wise:* $R \subseteq P$ *iff for all* $E \in Gen.R_E \subseteq P_E$*).*

Proposition 5.10. $\mathbb{S}_2$ *is a monotonic on the complete lattice of Gen-indexed families of binary relations over* $Obs_1 \times Obs_2$ *ordered by component-wise inclusion.*

By Tarski's fixpoint theorem $\mathbb{S}_2$ has the greatest fixpoint equal to the relativized simulation relation $(\leqslant)_{E \in Gen_{\mathcal{E}}}$ of Def. 5.8, which justifies its correctness.

Definition 5.11. *Consider three IOATSs: an environment* $\mathcal{E} = (Out, In, Gen, Obs, \overset{!}{\rightarrow}, \overset{?}{\rightarrow}, E^0)$ *and two systems* $\mathcal{S}_1 = (In, Out, Gen_1, Obs_1, \overset{!}{\rightarrow}_1, \overset{?}{\rightarrow}_1, s_1^0)$ *and* $\mathcal{S}_2 = (In, Out, Gen_2, Obs_2, \overset{!}{\rightarrow}_2, \overset{?}{\rightarrow}_2, s_2^0)$. *A Gen-indexed family of binary relations* $R : Gen \rightarrow \mathcal{P}(Obs_1 \times Obs_2)$ *constitutes a relativized bisimulation on observers of* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *iff* $(s_1, s_2) \in R_E$ *implies that:*

whenever $E \overset{i!}{\rightarrow} e \wedge e \overset{o?}{\rightarrow} E'$

then whenever $s_1 \overset{i?}{\rightarrow} S_1 \wedge S_1 \overset{o!}{\rightarrow} s_1'$

then for some S_2, s_2' *also* $s_2 \overset{i?}{\rightarrow} S_2 \wedge S_2 \overset{o!}{\rightarrow} s_2'$ *and* $(s_1', s_2') \in R_{E'}$,

and whenever $s_2 \overset{i?}{\rightarrow} S_2 \wedge S_2 \overset{o!}{\rightarrow} s_2'$

then for some S_1, s_1' *also* $s_1 \overset{i?}{\rightarrow} S_1 \wedge S_1 \overset{o!}{\rightarrow} s_1'$ *and* $(s_1', s_2') \in R_{E'}$.

*Let R be the largest of such families of relations ordered by component-wise
inclusion. We say that observers s_1, s_2 are equivalent with respect to a gen-
erator E, written $s_1 \sim_E s_2$, if $(s_1, s_2) \in R_E$. Systems $\mathcal{S}_1, \mathcal{S}_2$ are equivalent
in the context of $\mathcal{E}$, written $\mathcal{S}_1 \sim_\mathcal{E} \mathcal{S}_2$, if $s_1^0 \sim_{E^0} s_2^0$.*

The correctness of the construction follows in the same way as above. Just use the
function $\mathbb{B}_2$ instead of $\mathbb{S}_2$:

$$\mathbb{B}_2(R) = \lambda E. \ \{(s_1, s_2) \mid \forall i, o, e, E'. E \xrightarrow{i!} e \wedge e \xrightarrow{o?} E' \wedge$$

$$(\forall S_1, s_1'. \exists S_2, s_2'. s_1 \xrightarrow{i?} S_1 \wedge S_1 \xrightarrow{o!} s_1'$$

$$\Rightarrow s_2 \xrightarrow{i?} S_2 \wedge S_2 \xrightarrow{o!} s_2' \wedge (s_1', s_2') \in R_{E'}) \wedge$$

$$(\forall S_2, s_2'. \exists S_1, s_1'. s_2 \xrightarrow{i?} S_2 \wedge S_2 \xrightarrow{o!} s_2'$$

$$\Rightarrow s_1 \xrightarrow{i?} S_1 \wedge S_1 \xrightarrow{o!} s_1' \wedge (s_1', s_2') \in R_{E'})\} \qquad (5.7)$$

The two IOATSs on the left of Fig. 5.1, $\mathcal{M}$ and $\mathcal{I}$, represent systems,
while the two on the right, $\mathcal{E}_1$ and $\mathcal{E}_2$, model environments. White states
are observers and gray states are generators. The signature sets for the
systems are $In = \{i_1, i_2, i_3\}$ and $Out = \{o_1, o_2, o_3\}$. Assume that $\mathcal{M}$ is the
specification of the system, while $\mathcal{I}$ is an implementation. Under regular
simulation $\mathcal{I}$ does not conform to $\mathcal{M}$: $\mathcal{I} \not\leqslant \mathcal{M}$. $\mathcal{I}$ can produce o_1 in reaction
to i_2, which $\mathcal{M}$ does not allow. However $\mathcal{M}$ simulates $\mathcal{I}$ if executed in the
context of $\mathcal{E}_1$, because $\mathcal{E}_1$ cannot execute parts of the systems that differ:
$\mathcal{I} \leqslant_{\mathcal{E}_1} \mathcal{M}$. The environment $\mathcal{E}_2$ is more demanding than $\mathcal{E}_1$ as it can actually
detect that $\mathcal{I}$ does not conform, by producing i_2 and observing the reaction,
so $\mathcal{I} \not\leqslant_{\mathcal{E}_2} \mathcal{M}$. Also note that systems $\mathcal{M}$ and $\mathcal{I}$ are equivalent in environment
$\mathcal{E}_1$: $\mathcal{I} \sim_{\mathcal{E}_1} \mathcal{M}$, and not under $\mathcal{E}_2$: $\mathcal{I} \not\sim_{\mathcal{E}_2} \mathcal{M}$.

The notion of relativized simulation (bisimulation) was originally intro-
duced by Larsen in [69, 70] for ordinary labeled transition systems and used
as the basis for a compositional proof methodology [73]. An E-relativized
simulation between two systems s_1 and s_2 is closely related to an ordinary
simulation between the corresponding closed systems (E, s_1) and (E, s_2).
However, whereas the interactions of E with s_1 and s_2 are identical for rel-
ativized simulation (by its very definition), E may interact with s_1 and s_2
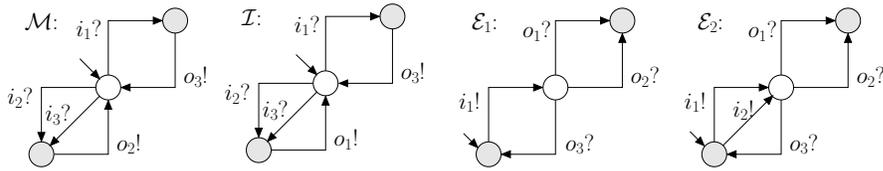
Figure 5.1: Systems $\mathcal{M}$ and $\mathcal{I}$ and environments $\mathcal{E}_1, \mathcal{E}_2$

in two radically different ways in order to establish a simulation between (E, s_1) and (E, s_2). In general, relativized simulation is a stronger relationship than simulation between closed systems; for deterministic environments the two notions coincide. Crucially, for our methodology, the notion of relativized simulation provides a simple (to state but not to prove) and elegant characterization of the discriminating power of environments (theorem 5.24) allowing for easy comparison and combination of environmental behavioral constraints.

5.2 Color-blind I/O-alternating Transition Systems

In the previous section we have presented a basic setup for specifying systems and environments together with their composition. We were able to state that two systems are equivalent with respect to a certain context if this context cannot activate their incompatible parts. However, in industrial development, it often happens that the environment cannot distinguish two systems, not because it makes incompatible parts unreachable, but because its ability to distinguish the *different outputs* it observes might be more or less limited depending on its actual state. For instance two outputs of the system may in some of its variants be connected to a single physical actuator. In such case the environment, being a model of the hardware in this case, will treat the two outputs as identical, allowing for powerful optimizations when generating code for this specific type of hardware. For this particular example, the distinguishing capability of the environment is clearly static and hence the specification of code optimization is realizable using simple process algebraic operations such as relabeling and hiding. However, in general environmental restrictions are dynamically changing (examples in chapter 6), and cannot be modeled using relabeling and hiding, which are both static operators and affect all transitions of the system regardless of the dynamics of environment.

To give a proper treatment of such situations we need to relax the equivalence of labels in relativized simulation (bisimulation), from equality to something weaker. We shall label observation transitions of environments with sets of inputs called *observation classes*. An observation transition can be taken in presence of any of the inputs in the set labeling it.

Definition 5.12. *A* color-blind *I/O alternating transition system is a tuple* $\mathcal{E} = (In, Out, Gen, Obs, \xrightarrow{!}, \xrightarrow{?}, E^0)$, *where In is a set of inputs, Out is a set of outputs, Gen denotes a finite set of generators, Obs denotes a finite set of color-blind observers,* $\xrightarrow{!} \subseteq Gen \times Out \times Obs$ *is the generation relation and* $\xrightarrow{?} \subseteq Obs \times \mathcal{P}(In) \times Gen$ *is the color-blind observation relation. The initial*

state is a generator: $E^0 \in Gen$.

A color-blind environment $\mathcal{E} = (In_{\mathcal{E}}, Out_{\mathcal{E}}, Gen_{\mathcal{E}}, Obs_{\mathcal{E}}, \overset{!}{\rightarrow}_{\mathcal{E}}, \overset{?}{\rightarrow}_{\mathcal{E}}, E)$ and a usual IOATS $\mathcal{S} = (In_{\mathcal{S}}, Out_{\mathcal{S}}, Gen_{\mathcal{S}}, Obs_{\mathcal{S}}, \overset{!}{\rightarrow}_{\mathcal{S}}, \overset{?}{\rightarrow}_{\mathcal{S}}, s)$ are compatible if they have matching signatures: $In_{\mathcal{E}} = Out_{\mathcal{S}} \wedge Out_{\mathcal{E}} = In_{\mathcal{S}}$. Since we only consider compatible systems and environments, we fix the meaning of the input and output, choosing the system's perspective. We will denote the set of inputs of the system by *In*. Consequently *In* is actually the set of outputs of the environment. Similarly *Out* is the set of outputs of the system but the set of inputs for the environment. Single input will be denoted by i, single outputs by o, and classes of outputs by capital O. We shall continue to write $E \overset{i!}{\rightarrow} e$ instead of $(E, i, e) \in \overset{!}{\rightarrow}$ and $e \overset{O?}{\rightarrow} E$ instead of $(e, O, E) \in \overset{?}{\rightarrow}$.

Composition is defined for compatible systems and environments. A compatible environment-system pair forms a closed system, which takes transitions synchronously. Synchronization is no longer equality based:

$$\frac{E \overset{i!}{\rightarrow} e \quad s \overset{i?}{\rightarrow} S}{(E, s) \rightarrow (e, S)} \quad \frac{S \overset{o!}{\rightarrow} s \quad e \overset{O?}{\rightarrow} E \quad o \in O}{(e, S) \rightarrow (E, s)} \tag{5.8}$$

We require that the observers in color-blind IOATS are deterministic and input enabled, so they deterministically react to every possible input. This implies that the observation classes on the transitions outgoing from a single state form a partitioning of inputs into equivalence classes. Formally for each observer e:

$$\forall O_1, O_2 \subseteq Out. \forall E_1, E_2 \in Gen. \ e \overset{O_1?}{\longrightarrow} E_1 \wedge e \overset{O_2?}{\longrightarrow} E_2$$

$$\Rightarrow O_1 \cap O_2 = \emptyset \vee (O_1 = O_2 \wedge E_1 = E_2)$$

$$\forall o \in Out. \exists O \subseteq Out. \exists E \in Gen. e \overset{O?}{\longrightarrow} E \wedge o \in O. \tag{5.9}$$

In addition we require that the generation relation is deterministic. For each generator E:

$$\forall i \in In. \forall e_1, e_2 \in Obs. E \overset{i!}{\rightarrow} e_1 \wedge E \overset{i!}{\rightarrow} e_2 \Rightarrow e_1 = e_2. \tag{5.10}$$

Note that determinism in the above sense does not limit the freedom of the environment in choosing inputs. It means that the input chosen uniquely determines the target state of the environment.

Consider a blind environment $\mathcal{B}$ with two states, a generator **B** and an observer **b**, which has a generation transition from **B** to **b** for every input i and a single observation transition from **b** to **B** labeled by *Out*. Intuitively $\mathcal{B}$ can execute all parts of the system, but does not really care about the responses it gets:

$$\forall i \in In. \ \mathbf{B} \overset{i!}{\rightarrow} \mathbf{b} \text{ and } \mathbf{b} \overset{Out?}{\longrightarrow} \mathbf{B} \ .$$

A perfect vision environment $\mathcal{V}$ consists of a generator $\mathbf{V}$ and an observer $\mathbf{v}$. $\mathcal{V}$ carefully observes all the outputs received from the system:

$$\forall i \in \mathit{In}.\ \mathbf{V} \xrightarrow{i!} \mathbf{v} \text{ and } \forall o \in \mathit{Out}.\ \mathbf{v} \xrightarrow{\{o\}?} \mathbf{V}\ .$$

In order to assert correctness of specialization of a system with respect to the possible execution scenarios, we need a notion of a context-dependent conformance between the original and the specialized system. We enrich our previous definition of relativized simulation and bisimulation to accommodate the color-blindness of environments.

Definition 5.13. *Let $\mathcal{E} = (\mathit{Out}, \mathit{In}, \mathit{Gen}, \mathit{Obs}, \xrightarrow{!}, \xrightarrow{?}, E^0)$ be a color-blind environment and $\mathcal{S}_1 = (\mathit{In}, \mathit{Out}, \mathit{Gen}_1, \mathit{Obs}_1, \xrightarrow{!}_1, \xrightarrow{?}_1, s_1^0)$ and $\mathcal{S}_2 = (\mathit{In}, \mathit{Out}, \mathit{Gen}_2, \mathit{Obs}_2, \xrightarrow{!}_2, \xrightarrow{?}_2, s_2^0)$ be two systems. A Gen-indexed family of binary relations $R : \mathit{Gen} \to \mathcal{P}(\mathit{Obs}_1 \times \mathit{Obs}_2)$ constitutes a relativized simulation iff $(s_1, s_2) \in R_E$ implies that*

$$\text{whenever } E \xrightarrow{i!} e \wedge e \xrightarrow{O?} E'$$

$$\text{then whenever } s_1 \xrightarrow{i?} S_1 \wedge S_1 \xrightarrow{o_1!} s_1' \wedge o_1 \in O$$

$$\text{then for some } S_2, o_2, s_2' \text{ also } s_2 \xrightarrow{i?} S_2 \wedge S_2 \xrightarrow{o_2!} s_2'$$

$$\wedge\, o_2 \in O \text{ and } (s_1', s_2') \in R_{E'}.$$

Let R be the largest of such families of relations ordered by component-wise inclusion. An observer s_2 simulates an observer s_1 in the context of generator E, written $s_1 \leqslant_E s_2$, if $(s_1, s_2) \in R_E$. An IOATS $\mathcal{S}_2$ simulates another IOATS $\mathcal{S}_1$ in the context of a compatible color-blind IOATS $\mathcal{E}$, written $\mathcal{S}_1 \leqslant_{\mathcal{E}} \mathcal{S}_2$, iff $s_1^0 \leqslant_{E^0} s_2^0$.

Let $\mathcal{S}_1$, $\mathcal{S}_2$ be IOATSs and $\mathcal{E}$ be a color-blind IOATS compatible with them (as in the above definition). Let $\mathbb{S}_3$ be an endofunction on a *Gen*-indexed families of binary relations of type $\mathit{Gen} \to \mathcal{P}(\mathit{Obs}_1, \mathit{Obs}_2)$ such that:

$\mathbb{S}_3(R) = R'$, where
$R' = \lambda E.\{(s_1, s_2) \mid \forall i, e, O, E'.\ \forall i, S_1, o_1, s_1'.\ \exists S_2, o_2.$

$$E \xrightarrow{i!} e \wedge e \xrightarrow{O?} E' \wedge s_1 \xrightarrow{i?} S_1 \wedge S_1 \xrightarrow{o_1!} s_1' \wedge o_1 \in O$$

$$\Rightarrow s_2 \xrightarrow{i?} S_2 \wedge S_2 \xrightarrow{o_2!} s_2' \wedge o_2 \in O \wedge (s_1', s_2') \in R'_E\}\ .$$

Proposition 5.14. *A Gen-indexed family of binary relations R constitutes a relativized simulation with respect to a color-blind IOATS iff $R \subseteq \mathbb{S}_3(R)$ (where inclusion is defined component-wise).*

Proposition 5.15. $\mathbb{S}_3$ *is a monotonic endofunction on the complete lattice of Gen-indexed famillies of binary relations over $Obs_1 \times Obs_2$ ordered by component-wise inclusion.*

It follows from Tarski's fixpoint theorem that $\mathbb{S}_3$ has the greatest fixpoint, which is equal to the relativized simulation relation of Def. 5.13 justifying the correctness of our definition.

Definition 5.16. *Let $\mathcal{E} = (Out, In, Gen, Obs, \xrightarrow{!}, \xrightarrow{?}, E^0)$ be a color-blind environment and $\mathcal{S}_1 = (In, Out, Gen_1, Obs_1, \xrightarrow{!}_1, \xrightarrow{?}_1, s_1^0)$ and $\mathcal{S}_2 = (In, Out, Gen_2, Obs_2, \xrightarrow{!}_2, \xrightarrow{?}_2, s_2^0)$ be two systems. A Gen-indexed family of binary relations $R : Gen \to \mathcal{P}(Obs_1 \times Obs_2)$ constitutes a relativized bisimulation iff $(s_1, s_2) \in R_E$ implies that*

$$whenever\ E \xrightarrow{i!} e \wedge e \xrightarrow{O?} E'$$

$$then\ whenever\ s_1 \xrightarrow{i?} S_1 \wedge S_1 \xrightarrow{o_1!} s_1' \wedge o_1 \in O$$

$$then\ for\ some\ S_2, o_2, s_2'\ also\ s_2 \xrightarrow{i?} S_2 \wedge S_2 \xrightarrow{o_2!} s_2'$$

$$\wedge o_2 \in O\ and\ (s_1', s_2') \in R_{E'}\ ,$$

$$and\ whenever\ s_2 \xrightarrow{i?} S_2 \wedge S_2 \xrightarrow{o_2!} s_2' \wedge o_2 \in O$$

$$then\ for\ some\ S_1, o_2, s_1'\ also\ s_1 \xrightarrow{i?} S_1 \wedge S_1 \xrightarrow{o_1!} s_1'$$

$$\wedge o_1 \in O\ and\ (s_1', s_2') \in R_{E'}\ .$$

Let R be the largest of such families of relations ordered by component-wise inclusion. Observers s_1, s_2 are equivalent in the context of generator E, written $s_1 \sim_E s_2$, if $(s_1, s_2) \in R_E$. Two IOATS $\mathcal{S}_1, \mathcal{S}_2$ are equivalent in the context of a compatible color-blind IOATS $\mathcal{E}$, written $\mathcal{S}_1 \sim_{\mathcal{E}} \mathcal{S}_2$, iff $s_1^0 \sim_{E^0} s_2^0$.

The correctness argument follows the standard pattern employing the $\mathbb{B}_3$ function:

$$\mathbb{B}_3(R) = \lambda E.\ \{(s_1, s_2) \mid \forall i, O, e, E'.E \xrightarrow{i!} e \wedge e \xrightarrow{O?} E' \wedge$$

$$(\forall o_1, S_1, s_1'. \exists S_2, s_2', o_2. s_1 \xrightarrow{i?} S_1 \wedge S_1 \xrightarrow{o_1!} s_1' \wedge o_1 \in O$$

$$\Rightarrow s_2 \xrightarrow{i?} S_2 \wedge S_2 \xrightarrow{o_2!} s_2' \wedge o_2 \in O \wedge (s_1', s_2') \in R_{E'}) \wedge$$

$$(\forall o_2, S_2, s_2'. \exists S_1, s_1', o_1. s_2 \xrightarrow{i?} S_2 \wedge S_2 \xrightarrow{o_2!} s_2' \wedge o_2 \in O$$

$$\Rightarrow s_1 \xrightarrow{i?} S_1 \wedge S_1 \xrightarrow{o_1!} s_1' \wedge o_1 \in O \wedge (s_1', s_2') \in R_{E'})\} \quad (5.11)$$

Color-blind IOATSs are a conservative extension of the usual (deterministic) IOATSs from the previous section. Any deterministic IOATS can

be translated into a color-blind IOATS, by turning labels on observation transitions into singleton sets. In the same sense our new relativized simulation (bisimulation) is a conservative extension of the one from the previous section (applied to deterministic IOATSs).

Even though we initially postulated that execution contexts usually do not exercise all possible traces of the system, we will now require that environments can always produce all the possible system inputs from *In*. This requirement surprisingly does not defeat our initial goal. We can direct all the transitions producing impossible system inputs to the observer **b** and embed the blind environment $\mathcal{B}$ in every environment model. Instead of specifying that the environment cannot produce i, we state that i can be produced, but the environment does not care about the behavior of the system afterward. Proposition 5.17 formally states that the system's behavior is irrelevant once the environment turns blind.

Proposition 5.17. *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two IOATSs of the same signature and $\mathcal{B}$ be a color-blind environment compatible with them (defined as above). For any two observers $s_1 \in Obs_1$ and $s_2 \in Obs_2$ it holds that $s_1 \sim_{\mathbf{B}} s_2$ and consequently $\mathcal{S}_1 \leqslant_{\mathcal{B}} \mathcal{S}_2$.*

Proof. It suffices to show that $R = \lambda B. \, Obs_1 \times Obs_2$ is a **B**-relativized bisimulation relation, which follows from input-enabledness of $\mathcal{S}_1$ and $\mathcal{S}_2$ and from the fact that the observation class of the only observation transition of $\mathcal{B}$ is the entire *Out* set. $\square$

Figure 5.2 presents two examples of environments that are compatible with the systems of Fig. 5.1. Transitions from generators to the blind observer **b** have been omitted, which will be our usual practice. There is one such transition for each input–generator pair, for which the transition is not drawn. Observe that the model $\mathcal{M}$ simulates the implementation $\mathcal{I}$ in the environment $\mathcal{F}_1$ ($\mathcal{I} \leqslant_{\mathcal{F}_1} \mathcal{M}$) not due to the fact that $\mathcal{F}_1$ is not able to exercise the differing parts of the systems, as was the case for $\mathcal{E}_1$, but because $\mathcal{F}_1$ cannot distinguish between the outputs (o_1, o_2) produced by $\mathcal{I}$ and $\mathcal{M}$. The almost identical color-blind environment $\mathcal{F}_2$ distinguishes $\mathcal{I}$ and $\mathcal{M}$, by observing the outputs o_1 and o_2 with two separate transitions. In fact $\mathcal{I} \sim_{\mathcal{F}_1} \mathcal{M}$ and $\mathcal{I} \not\sim_{\mathcal{F}_2} \mathcal{M}$.
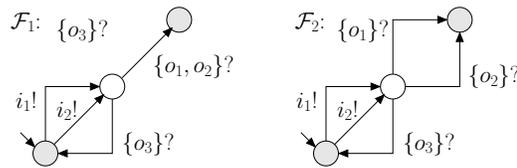
Figure 5.2: Color-blind environments $\mathcal{F}_1, \mathcal{F}_2$ compatible with systems $\mathcal{M}$ and $\mathcal{I}$: $\mathcal{I} \leqslant_{\mathcal{F}_1} \mathcal{M}$, $\mathcal{I} \not\leqslant_{\mathcal{F}_2} \mathcal{M}$, $\mathcal{I} \sim_{\mathcal{F}_1} \mathcal{M}$ and $\mathcal{I} \not\sim_{\mathcal{F}_2} \mathcal{M}$.

Relativized simulation and bisimulation are weaker than their nonrelativized versions:

Proposition 5.18. *Let $\mathcal{S}_1 = (In, Out, Gen_1, Obs_1, \xrightarrow{!}_1, \xrightarrow{?}_1, s_1^0)$ and $\mathcal{S}_2 = (In, Out, Gen_2, Obs_2, \xrightarrow{!}_2, \xrightarrow{?}_2, s_2^0)$ be two IOATSs and $\mathcal{E} = (Out, In, Gen, Obs, \xrightarrow{!}, \xrightarrow{?}, E^0)$ be a color-blind IOATS: Let s_1 and s_2 be any two observers of $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively and E be an arbitrary generator of $\mathcal{E}$: $E \in Gen$. Then $s_1 \leqslant s_2$ implies $s_1 \leqslant_E s_2$, and $\mathcal{S}_1 \leqslant \mathcal{S}_2$ implies $\mathcal{S}_1 \leqslant_{\mathcal{E}} \mathcal{S}_2$. Similarly $s_1 \sim s_2$ implies $s_1 \sim_E s_2$ and $\mathcal{S}_1 \sim \mathcal{S}_2$ implies $\mathcal{S}_1 \sim_{\mathcal{E}} \mathcal{S}_2$.*

Proof. First show that $R_1 = \lambda E.\{(s_1, s_2) \mid s_1 \leqslant s_2\}$ constitutes an $\mathcal{E}$-relativized simulation on observers of $\mathcal{S}_1$ and $\mathcal{S}_2$ (follows directly from definition 5.13). Then show that $R_2 = \lambda E.\{(s_1, s_2) \mid s_1 \sim s_2\}$ constitutes an $\mathcal{E}$-relativized bisimulation on observers of $\mathcal{S}_1$ and $\mathcal{S}_2$. The results on the IOATS level follow as immediate corollaries. $\qquad\square$

Moreover the perfect vision environment $\mathcal{V}$ is the most discriminating one. It can distinguish any systems different up to the usual simulation (so relativized simulation is a conservative extension of def. 5.2 for deterministic environments).

Proposition 5.19. *For any two observers s_1, s_2 of systems $\mathcal{S}_1$ and $\mathcal{S}_2$, and the generator $\mathbf{V}$ of the color-blind perfect-vision environment $\mathcal{V}$ compatible with $\mathcal{S}_1$ and $\mathcal{S}_2$: $s_1 \sim s_2 \iff s_1 \sim_V s_2$ and $\mathcal{S}_1 \sim \mathcal{S}_2 \iff \mathcal{S}_1 \sim_{\mathcal{V}} \mathcal{S}_2$. Similarly $s_1 \leqslant s_2 \iff s_1 \leqslant_{\mathbf{V}} s_2$ and $\mathcal{S}_1 \leqslant \mathcal{S}_2 \iff \mathcal{S}_1 \sim_{\mathcal{V}} \mathcal{S}_2$.*

Proof. First show that $R = \lambda V.\{(s_1, s_2) \mid s_1 \sim s_2\}$ is a $\mathcal{V}$-relativized bisimulation on observers of $\mathcal{S}_1$ and $\mathcal{S}_2$, which follows directly from the definition. In order to prove the other direction we need to show that $\sim_V$ constitutes a usual bisimulation on observers of $\mathcal{S}_1$ and $\mathcal{S}_2$. Take s_1 and s_2 such that $s_1 \sim_V s_2$. Considering the definition of simulation (definition 5.2) take any two transitions of $\mathcal{S}_1$ such that $s_1 \xrightarrow{i?} S_1 \wedge S_1 \xrightarrow{o!} s_1'$. Note that by definition of $\mathcal{V}$ we have $V \xrightarrow{i!} v$ and $v \xrightarrow{\{i\}?} V$. And then because of input enabledness and $s_1 \sim_V s_2$ we can observe that $s_2 \xrightarrow{i?} S_2$ and $S_2 \xrightarrow{o!} s_2'$ and $s_1' \sim_V s_2'$ (and vice-versa). So $\sim_{\mathcal{V}}$ fulfills definition 5.2 and constitutes a bisimulation on observers of $\mathcal{S}_1$ and $\mathcal{S}_2$. The result on the IOATS level follows as a direct corollary. The proof of the simulation case is similar (and simpler). $\qquad\square$

With the above propositions we have hinted at the notion of *discrimination*—the ability of distinguishing systems from each other.

Definition 5.20. *A color-blind environment $\mathcal{F}$ is more discriminating than a color-blind environment $\mathcal{E}$, written $\mathcal{E} \sqsubseteq \mathcal{F}$, iff $\mathcal{F}$ distinguishes more processes than $\mathcal{E}$. Formally $\mathcal{E} \sqsubseteq \mathcal{F}$ iff $\forall \mathcal{S}_1, \mathcal{S}_2.\mathcal{S}_1 \leqslant_{\mathcal{F}} \mathcal{S}_2 \Rightarrow \mathcal{S}_1 \leqslant_{\mathcal{E}} \mathcal{S}_2$.*

Our blind environment $\mathcal{B}$ is the least discriminating in the sense that it is unable to distinguish any two systems from each other by means of relativized simulation. The perfect vision environment $\mathcal{V}$ is the most discriminating (by propositions 5.18 and 5.19 as no other environment can distinguish more systems).

The notion of discrimination will soon prove to be fundamental in our developments. In the next section we shall use it to design composition operators for behavioral properties. Those compositions shall later form the fundamental part of our tool for designing product line architectures. Unfortunately our definition of the discrimination preorder is rather abstract. Due to quantification over all possible systems, it is not possible to verify it mechanically. To remedy this obstacle we introduce a new preorder on environments: a simulation for color-blind environments.

Definition 5.21. *Let* $\mathcal{E} = (Out, In, Gen_{\mathcal{E}}, Obs_{\mathcal{E}}, \xrightarrow{!}_{\mathcal{E}}, \xrightarrow{?}_{\mathcal{E}}, E^0)$ *and* $\mathcal{F} = (Out,$ $In, Gen_{\mathcal{F}}, Obs_{\mathcal{F}}, \xrightarrow{!}_{\mathcal{F}}, \xrightarrow{?}_{\mathcal{F}}, F^0)$ *be color-blind environments. A pair of binary relations,* $R_1 \subseteq Gen_{\mathcal{E}} \times Gen_{\mathcal{F}}$ *and* $R_2 \subseteq Obs_{\mathcal{F}} \times Obs_{\mathcal{E}}$, *constitutes a simulation between states of color-blind IOATSs iff* $(E, F) \in R_1$ *implies that*

$$\text{whenever } E \xrightarrow{i!} e \text{ then for some } f \text{ also } F \xrightarrow{i!} f \text{ and } (f, e) \in R_2 \ ,$$

and $(f, e) \in R_2$ *implies that whenever* $f \xrightarrow{O_f?} F$

$$\text{then for some } O_e, E \text{ also } e \xrightarrow{O_e?} E \text{ and } O_f \subseteq O_e \text{ and } (E, F) \in R_1 \ .$$

Let (R_1, R_2) *be the largest such pair of relations (ordered by point-wise inclusion). A generator* F *simulates a generator* E, *written* $E \leqslant F$, *iff* $(E, F) \in R_1$. *An observer* e *simulates an observer* f, *written* $f \leqslant e$, *iff* $(f, e) \in R_2$. *An environment* $\mathcal{F}$ *simulates an environment* $\mathcal{E}$, *written* $\mathcal{E} \leqslant \mathcal{F}$, *iff* $E^0 \leqslant F^0$.

Consider two color-blind environments $\mathcal{E} = (Out, In, Gen_{\mathcal{E}}, Obs_{\mathcal{E}}, \xrightarrow{!}_{\mathcal{E}}, \xrightarrow{?}_{\mathcal{E}}, E^0)$ and $\mathcal{F} = (Out, In, Gen_{\mathcal{F}}, Obs_{\mathcal{F}}, \xrightarrow{!}_{\mathcal{F}}, \xrightarrow{?}_{\mathcal{F}}, F^0)$. Let $\mathbb{S}_4$ be an endofunction on pairs of binary relations such that:

$\mathbb{S}_4 : \mathcal{P}(Gen_{\mathcal{E}} \times Gen_{\mathcal{F}}) \times \mathcal{P}(Obs_{\mathcal{F}} \times Obs_{\mathcal{E}}) \rightarrow \mathcal{P}(Gen_{\mathcal{E}} \times Gen_{\mathcal{F}}) \times \mathcal{P}(Obs_{\mathcal{F}} \times Obs_{\mathcal{E}})$
$\mathbb{S}_4(R_1, R_2) = (R_1', R_2')$, where

$R_1' = \{(E, F) \mid \forall i, e. \ \exists f. \ E \xrightarrow{i!} e \Rightarrow F \xrightarrow{i!} f \wedge (f, e) \in R_2\}$

$R_2' = \{(f, e) \mid \forall O_f, F. \ \exists O_e, E. \ f \xrightarrow{O_f?} F \Rightarrow e \xrightarrow{O_e?} E \wedge (E, F) \in R_1\} \ .$

Proposition 5.22. *A pair of relations* (R_1, R_2) *constitutes a simulation between states of two color-blind IOATSs iff* $(R_1, R_2) \subseteq \mathbb{S}_4(R_1, R_2)$, *where inclusion is intepreted point-wise.*

Proposition 5.23. $\mathbb{S}_4$ *is a monotonic endofunction on the complete lattice of pairs of binary relations over generators and observers (accordingly), ordered by point-wise inclusion.*

By Tarski's fixpoint theorem $\mathbb{S}_4$ has the greatest fixpoint equal to the simulation relation $\leqslant$ of Def. 5.21, which justifies the correctness of our definition.

The simulation preorder can be checked mechanically for finite state systems using state exploration techniques [20, 9]. Thanks to the following central result, these techniques can also be used to verify discrimination properties, as discrimination ordering and simulation ordering coincide for our environments:

Theorem 5.24. *For any two color-blind environments $\mathcal{E}$ and $\mathcal{F}$:*

$$\mathcal{E} \sqsubseteq \mathcal{F} \text{ iff } \mathcal{E} \leqslant \mathcal{F} \ .$$

Let us begin the proof with introducing an auxiliary definition of discrimination on the state level:

Definition 5.25. *Let $\mathcal{E}$ and $\mathcal{F}$ be two color-blind environments having the same signatures and let $E \in Gen_{\mathcal{E}}$, $F \in Gen_{\mathcal{F}}$ be two generators. We say that F is more discriminating than E, written $E \sqsubseteq F$, if for any two observers S_1, S_2 of any two systems $\mathcal{S}_1, \mathcal{S}_2$ compatible with $\mathcal{E}$ and $\mathcal{F}$ it holds that $S_1 \leqslant_F S_2$ implies $S_1 \leqslant_E S_2$.*

Lemma 5.26. *Let $\mathcal{E}$ and $\mathcal{F}$ be two color-blind environments having the same signatures. For any two color-blind generators $E \in Gen_{\mathcal{E}}$ and $F \in Gen_{\mathcal{F}}$, with the same I/O signature it holds that $E \leqslant F \Rightarrow E \sqsubseteq F$.*

Proof. Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two systems compatible with $\mathcal{E}$ and $\mathcal{F}$. Assume that $E \leqslant F$. We need to show that for any two observers s_1, s_2 of $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively $s_1 \leqslant_F s_2$ implies $s_1 \leqslant_E s_2$, or in other words $(\leqslant_F) \subseteq (\leqslant_E)$. We shall show this in two steps, first introducing a $Gen_{\mathcal{E}}$-indexed family of relations R such that $(\leqslant_F) \subseteq R_E$, and then arguing that R_E is an $\mathcal{E}$-relativized simulation, which implies that it is included in relativized simulation component-wise, so $R_E \subseteq (\leqslant_E)$ Let us define R as follows:

$$R = \lambda E. \ \{(s_1, s_2) \mid \exists F' \in Gen_{\mathcal{F}}.E \leqslant F' \wedge s_1 \leqslant_{F'} s_2\} \ . \tag{5.12}$$

Clearly $(\leqslant_F) \subseteq R_F$, since $E \leqslant F$. Now take s_1, s_2 and E' such that $(s_1, s_2) \in R_{E'}$. Let $E' \xrightarrow{i!} e'$ and $e' \xrightarrow{O_e?} E''$ and $s_1 \xrightarrow{i?} S_1$ and $S_1 \xrightarrow{o_1!} s_1'$ and $o_1 \in O_e$. We need to find S_2, o_2, s_2' such that $s_2 \xrightarrow{i?} S_2$ and $S_2 \xrightarrow{o_2!} s_2'$ and $o_2 \in O_e$ and $(s_1', s_2') \in R_{E''}$. But since $(s_1, s_2) \in R_{E'}$ there must exists F' such that $s_1 \leqslant_{F'} s_2$ and $E \leqslant F'$. The latter means that there exist f', O_f, F'' such that $F' \xrightarrow{i!} f'$ and $f' \xrightarrow{O_f?} F''$ and $o_1 \in O_f \subseteq O_e$ and $E'' \leqslant F''$, which combined with the former implies that $s_2 \xrightarrow{i?} S_2$ and $S_2 \xrightarrow{o_2!} s_2'$ and $o_2 \in O_f \subseteq O_e$. It remains to be shown that $(s_1', s_2') \in R_{E''}$, which follows from the definition of R, as $E'' \leqslant F''$. $\square$
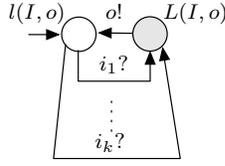
Figure 5.3: A looping system $\mathcal{L}(I, o)$, for $I = \{i_1, \ldots, i_k\}$

Lemma 5.27. *Let $\mathcal{E}$ and $\mathcal{F}$ be two color-blind environments having the same I/O signature. If E and F are generators of $\mathcal{E}$ and $\mathcal{F}$ respectively, then*

$$E \sqsubseteq F \Rightarrow E \leqslant F \ .$$

As a preparation for the proof let us define a looping system $\mathcal{L}(I, o)$. Let I be a set of inputs, $I \subseteq In$, and o be a single output, $o \in Out$. Then let $L(I, o)$ denote a generator and $l(I, o)$ denote an observer. In $\mathcal{L}(I, o)$ there is an observation transition from $l(I, o)$ to $L(I, o)$ for every $i \in I$ and a single generation transition $L(I, o) \xrightarrow{o!} l(I, o)$.

Proof. We shall prove the contrapositive instead, i.e for all $\mathcal{E}, \mathcal{F}$ and their generators E, F if $E \not\leqslant F$ then also $E \not\sqsubseteq F$, so there exist systems $\mathcal{S}_1, \mathcal{S}_2$ and their observers s_1, s_2 such that $s_1 \leqslant_F s_2$ but $s_1 \not\leqslant_E s_2$.

Since $E \not\leqslant F$ then there exists n such that E cannot cover the observation class of F at the nth observation step. We shall proceed by induction on n.

1°. if $n = 1$ then there exist input i_k, observation class O_f and observers e, f such that $E \xrightarrow{i_k!} e$ and $F \xrightarrow{i_k!} f$ and $f \xrightarrow{O_f!} F'$, but for all transitions outgoing from e, $e \xrightarrow{O_e?} E'$, we have that $O_f \not\subseteq O_e$. Because of this and the fact that the observation classes of e form a partitioning of Out (see (5.9)), there exist two distinct observation classes O'_e, O''_e of e, such that $O'_e \cap O_f \neq \emptyset$ and $O''_e \cap O_f \neq \emptyset$. Let o' be an arbitrary element from $O'_e \cap O_f$ and similarly $o'' \in O''_e \cap O_f$. We shall now construct our two systems $\mathcal{S}_1$ and $\mathcal{S}_2$. The idea is that the first steps of $\mathcal{S}_1$ and $\mathcal{S}_2$ differ insufficiently to be distinguished by f, but sufficiently for e to distinguish them. In the subsequent steps both systems behave identically. Let $\mathcal{S}_1$ just consist of $l(In, o')$ and $\mathcal{S}_2$ be as on Fig. 5.4. It is easy to observe that $s_1 \leqslant_F s_2$, but $s_1 \not\leqslant_E s_2$, which finishes the proof for $n = 1$.

2°. Inductive step. Now consider that $n > 1$ and E simulates F on all traces shorter than n. In a similar manner as above we would like to construct two systems which violate E-relativized simulation in the nth step on the very trace, on which E and F disagree. On all other traces of length n, and all longer traces they should behave identically. It should be intuitively visible that a construction similar to the one presented above is applicable.

More formally our new systems can be constructed inductively. Consider the prefixes of the execution witnessing $E \not\leqslant F$: $E \xrightarrow{i_k!} e, e \xrightarrow{O_e?} E'$ and $F \xrightarrow{i_k!} f$, $f \xrightarrow{O_f?} F'$. Since $n > 1$, $O_f \subseteq O_e$ and there exists a trace witnessing violation of simulation between E' and F' in $n - 1$ steps. By the induction hypothesis there
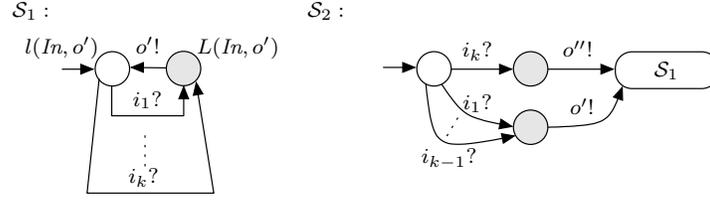
Figure 5.4: Counter example systems $\mathcal{S}_1$ and $\mathcal{S}_2$ drawn assuming that $In = \{i_1, \ldots, i_k\}$. Note that the proof does not rely on In being finite; k is just a notational convention, to ease the presentation.
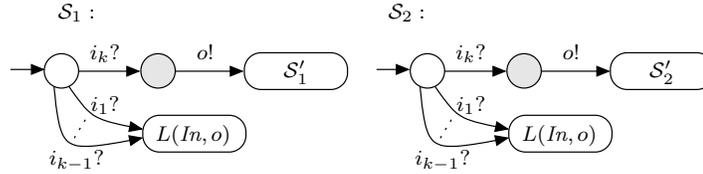
Figure 5.5: Systems $\mathcal{S}_1$ and $\mathcal{S}_2$ created in the inductive step of the proof of lemma 5.27. Again the drawing assumes that $In = \{i_1, \ldots, i_k\}$, but finiteness of In is rather a notational convention than an actual need.

exist systems $\mathcal{S}'_1$ and $\mathcal{S}'_2$ and states thereof S'_1 and S'_2 such that $S'_1 \leqslant_F S'_2$ and $S'_1 \not\leqslant_E S'_2$. We create a new pair of systems $\mathcal{S}_1$ and $\mathcal{S}_2$ by adding new initial observers s_1, s_2 and generators S_1, S_2 with transitions $s_1 \xrightarrow{i_k?} S_1, S_1 \xrightarrow{o!} s'_1$ and $s_2 \xrightarrow{i_k?} S_2$, $S_2 \xrightarrow{o!} s'_2$, where $o \in O_f \subseteq O_e$ and s'_1 and s'_2 are the initial states of $\mathcal{S}'_1$ and $\mathcal{S}'_1$. Both for s_1 and s_2 we also add transitions for all inputs different than i_k to $l(In, o)$. See Fig. 5.5. It is not hard to see that $s_1 \leqslant_F s_2$, but $s_1 \not\leqslant_E s_2$, which finishes the proof. □

Theorem 5.24 follows as a corollary from the above two lemma lifted to the IOATS level.

5.3   Composition of Behavioral Properties

The environment model can be used as an additional source of information for an optimizing code generator. Typical code generators do not use any information about the environment, assuming that the model is combined with the perfect vision environment $\mathcal{V}$. Another extreme would be a code generator requiring a precise model of the environment—definitely a significant burden for many engineers. To remedy the problem we propose light-weight, composable, partial specifications of environments. Examples of such properties could be: that certain events always come interleaved (like on/off switch), or that there is some causality between a certain input and

output (a timer only timeouts after it has been started). Each of such properties can be expressed as a simple color-blind environment IOATS. Our goal now is to show how such partial descriptions can be composed.

As said before, every observer e of a color-blind IOATS induces a partitioning of *Out* into observation classes. Let us denote this partitioning by P_e. The set of all equivalence relations (and hence the set of all partitionings) over *Out*, ordered by inclusion, forms a complete lattice. Consequently for any set of partitionings $\{P_k\}_{k \in L}$ there exist the greatest lower bound $\bigsqcap_{k \in L} P_k$, which is the coarsest partitioning finer than any of P_k and the least upper bound $\bigsqcup_{k \in L} P_k$, which is the finest partitioning coarser than all P_k.

The composition is defined for environments with the same I/O signatures. We consider two kinds of composition: a sum and a product. Sums intuitively correspond to disjunction of properties (or sums in CCS [94]). Products correspond to conjunctions (or synchronous composition in CSP [52]).

$$\frac{E_1 \xrightarrow{i!} e_1 \quad \dots \quad E_n \xrightarrow{i!} e_n}{\sum_{k=1}^{n} E_k \xrightarrow{i!} \prod_{k=1}^{n} e_k} \text{ SG} \tag{5.13}$$

$$\frac{E_1 \xrightarrow{i!} e_1 \quad \dots \quad E_n \xrightarrow{i!} e_n}{\prod_{k=1}^{n} E_k \xrightarrow{i!} \sum_{k=1}^{n} e_k} \text{ PG} \tag{5.14}$$

$$\frac{O \in \prod_{k=1}^{n} P_{e_k} \quad \mathbb{E} = \{E | \exists 1 \leq k \leq n. \exists O' \subseteq Out.e_k \xrightarrow{O'?} E \wedge O \subseteq O'\}}{\prod_{k=1}^{n} e_k \xrightarrow{O?} \sum \mathbb{E}} \text{ PO} \tag{5.15}$$

$$\frac{O \in \bigsqcup_{k=1}^{n} P_{e_k} \quad \mathbb{E} = \{E | \exists 1 \leq k \leq n. \exists O' \subseteq O.e_k \xrightarrow{O'?} E\}}{\sum_{k=1}^{n} e_k \xrightarrow{O?} \prod \mathbb{E}} \text{ SO} \tag{5.16}$$

The result of a composition is a well-formed color-blind IOATS (consisting of the same signature as the composed parts, and all states reachable by the above given rules). This can be argued using rule induction, checking that each generator created is deterministic and input-enabled, and that observation classes on all transitions are deterministic (non-overlapping) and non-blocking (complete). The sum is naturally lifted to the entire IOATSs with the same signatures.

The first two rules, for the sum of generators (SG) and for the product of generators (PG) are very simple, due to the determinism and input-enabledness of our generators. All generators in $\{E_k\}_{k=1}^{n}$ can generate any

$i \in In$. The composition is synchronous: all environments are taking the step simultaneously. From the system's perspective a single i is generated.

The observer rules are more complex, due to the determinisation performed on-the-fly. Consider the product of observers (PO) first. The observation classes O of the composed environment will be finer than observation classes of any of the composed processes. Whenever any o is observed by the result of the composition we advance to the state $\mathbb{E}$ composed of states reachable by o from all e_k's. Since O is finer than some class in any of these observers there is always exactly n such generators to be reached in $\mathbb{E}$.

Dually in the sum of observers (SO) observational classes are coarser than classes of any of the composed processes. The transition relation follows to those generators that can be reached by any output belonging to such an extended class. The size of $\mathbb{E}$ can be bigger than the number of original observers n.

Our compositions enjoy this essential property:

Theorem 5.28. *The sum of environments $\{\mathcal{E}\}_{k=1}^{n}$ is the least environment, which simulates all summands. The product of environments $\{\mathcal{E}\}_{k=1}^{n}$ is the greatest environment, which is simulated by all the factors.*

Proof. We shall show the theorem on the state level (the result on the IOATSs level follows directly). First show that $\forall k = 1 \ldots n.\ E_k \leqslant \sum_{k=1}^{n} E_k$. This is in fact the case because the product of observers creates observation classes which are always subsets of original classes (partitioning of the original classes). More formally it can be argued by showing that the pair of relations (R_1, R_2), defined as below forms a simulation on environments:

$$R_1 = \left\{ \left(E_j, \sum_{k \in I} E_k \right) \mid \text{for any finite } I \text{ and generators } \{E_k\}_{k \in I} \text{ and } j \in I \right\}$$

$$R_2 = \left\{ \left(\prod_{k \in I} e_k, e_j \right) \mid \text{for any finite } I \text{ and observers } \{e_k\}_{k \in I} \text{ and } j \in I \right\} \quad (5.17)$$

It remains to show that for all such generators F that $\forall k = 1 \ldots n.\ E_k \leqslant F$, it holds that also $\sum_{k=1}^{n} E_k \leqslant F$. This in turn is achieved by showing that the pair of relations (R_3, R_4) forms a simulation on environments, where:

$$R_3 = \left\{ \left(\sum_{k \in I} E_k, F \right) \mid \forall \text{ finite } I.\ \forall \{E_k\}_{k \in I}.\ \forall F.\ \forall k \in I.E_k \leqslant F \right\}$$

$$R_4 = \left\{ \left(f, \prod_{k \in I} e_k \right) \mid \forall \text{ finite } I.\ \forall \{e_k\}_{k \in I}.\ \forall f.\forall k \in I.\ f \leqslant e_k \right\} \quad (5.18)$$

The proof of the case for products of generators is dual. $\qquad\square$

Since discrimination and simulation coincide (theorem 5.24) this property could be rewritten using $\sqsubseteq$ instead of $\leqslant$: *The sum of environments is the*
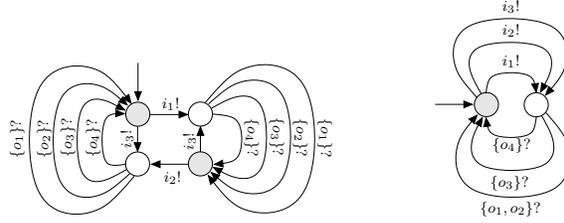
Figure 5.6: Properties *Interleave i_1 i_2* and *Equiv o_1 o_2* as environments. $In = \{i_1, i_2, i_3\}$, $Out = \{o_1, \dots, o_4\}$.
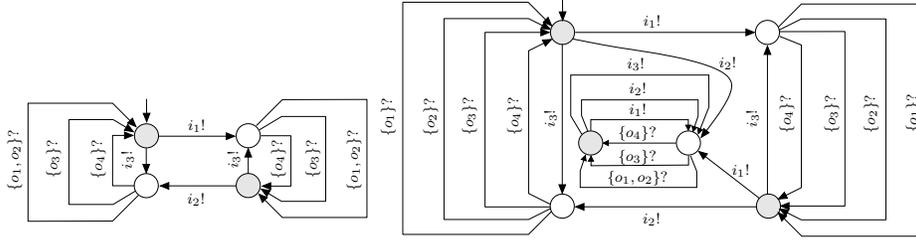
Figure 5.7: Product (left) and sum (right) of environments of Fig. 5.6. The product can only generate what both of the factors could generate and distinguish only what both of them could distinguish. The sum can generate what any of the summands could generate and observe what any of them could observe. In particular o_1 and o_2 are distinguished in the traces for which the *Interleave* property is preserved and not otherwise.

least discriminating environment, more discriminating than each of the summands. The product is the most discriminating environment, less discriminating than each of the factors. The classes of environments defined by two-way simulation ($\lesssim$) form a complete lattice with the discrimination ordering. All environments in a single class are equally discriminating. The blind environment $\mathcal{B}$ belongs to the bottom class in this lattice, while the perfect vision environment $\mathcal{V}$ is a member of the top class. A sum of two environments is an environment that belongs to the least upper bound class of classes containing the summands. Similarly the product is an environment that belongs to the greatest lower bound class of classes containing the factors. With these properties our composition operators can become the basis for disjunction and conjunction of behavioral properties in the specification language for environments.

As an example, Figure 5.6, consider two color-blind environments. First, *Interleave i_1 i_2* saying that two inputs i_1 and i_2 always alternate. Another interesting property is that two given outputs never can be distinguished by the environment: *Equiv o_1 o_2*. Both properties are defined conservatively. *Interleave* does not restrict anything in the ability of distinguishing outputs

or producing the third input i_3. *Equiv* does not restrict the way in which inputs are produced, only saying that o_1 and o_2 are always equivalent. These two environments could be combined in the following ways:

$$\llbracket Interleave\ i_1\ i_2 \land Equiv\ o_1\ o_2 \rrbracket = \llbracket Interleave\ i_1\ i_2 \rrbracket \times \llbracket Equiv\ o_1\ o_2 \rrbracket \quad (5.19)$$

$$\llbracket Interleave\ i_1\ i_2 \lor Equiv\ o_1\ o_2 \rrbracket = \llbracket Interleave\ i_1\ i_2 \rrbracket + \llbracket Equiv\ o_1\ o_2 \rrbracket \quad (5.20)$$

The first environment (conjunction) is guaranteed to be the most discriminating environment less discriminating than both operands. It will only be able to produce inputs in interleaved fashion and always consider o_1 and o_2 equivalent. It can be used as a specification for optimization of a reactive system, which is supposed to work in the environment exhibiting both properties. The second environment (disjunction) is the least discriminating environment more discriminating than both operands. It can be used as a specification for optimization of a reactive system, which is supposed to operate correctly both in an environment exhibiting the interleave property and in the environment exhibiting the equivalent property. Figure 5.7 demonstrates the results of both compositions.

5.4   Equivalence vs Refinement

A refinement relation states that one system implements parts of another system's functionality. An equivalence relation states that two systems implement exactly the same behavior in some sense. Intuitively refinement relations should serve the purpose of specialization better than equivalence. By their very definition they state requirements for systems, which can be seen as specializations/optimizations of other systems. At the same time equivalences seem to require preservation of rich nondeterminism between a model and its implementation (specialized variant), which is highly inconvenient for code generators. A compiler writer normally wants to choose one way of implementing the underspecified behavior, instead of preserving all, often exponentially many, possibilities.

Unfortunately many simple refinement relations do not preserve interesting properties of the system, most notably deadlock freeness. In this section we study preservation of deadlock freeness by relativized simulation, two-way relativized simulation and relativized bisimulation for IOATSs, emphasizing one strength of color-blindness, namely that color-blind relativized bisimulation not only behaves well (preserves deadlock properties), but also allows applications typical for refinement relations.

An execution of a given IOATS $\mathcal{S}$ is a sequence of transitions that can be taken according to the transition relations of $\mathcal{S}$.

Definition 5.29. *Let* $\mathcal{S} = (In,\ Out,\ Gen,\ Obs, \xrightarrow{!}, \xrightarrow{?}, s^0)$. *An execution from*

observer s_0 is a sequence of transitions:

$$s_0 \xrightarrow{i_1?} s_1 \wedge s_1 \xrightarrow{o_2!} s_2 \wedge s_2 \xrightarrow{i_3?} \ldots \xrightarrow{l_{n-1}} s_n \ldots \quad ,$$

such that all $s_i \in Obs \cup Gen$, where all $i_k \in In$ and all $o_k \in Out$. We say that a given state $s \in Gen \cup Obs$ is reachable in $\mathcal{S}$, if there exists an execution from s^0 containing s.

Note that if two states s_1, s_2 are in a simulation relation $(s_1 \leqslant s_2)$, then all executions of s_1 can be matched by s_2 (meaning that sequences of I/O labels extracted from executions are the same).

Let $\perp$ be a distinct member of *Out*, denoting an *empty output*. We say that an IOATS can *deadlock*, if there exists an execution such that after a finite number of steps, it will generate $\perp$ forever, no matter what inputs are arriving from the environment.

Definition 5.30. *We say that a given state s of an IOATS $\mathcal{S}$ is a deadlock state if all outputs of all possible executions from s are $\perp$. We say that $\mathcal{S}$ can deadlock if it has a reachable deadlock state.*

An IOATS that cannot deadlock is said to be *deadlock-free*. Deadlock freeness is one of the fundamental requirements usually enforced on high reliability software. It can be easily shown that simulation does not preserve deadlock freeness, i.e. it is possible that a deadlock-free system $\mathcal{S}_2$ is refined by a deadlocking system $\mathcal{S}_1$. Figure 5.8 presents a relevant counter example: $\mathcal{S}_1 \leqslant \mathcal{S}_2$, but $\mathcal{S}_1$ can deadlock, while $\mathcal{S}_2$ is deadlock free.

The property of deadlock freeness can be relativized in a straightforward way. A system $\mathcal{S}$ is *deadlock-free relative to an environment $\mathcal{E}$* (non color-blind), if it cannot deadlock, while executed in $\mathcal{E}$, i.e. there is no reachable observer state, such that any execution starting in it will only produce $\perp$ as a reply to inputs produced by $\mathcal{E}$.

Definition 5.31. *An execution of observer s_0 relative to a generator E_0 is a sequence of pairs of transitions:*

$$(s_0, E_0) \xrightarrow{i_1} (S_1, e_1) \wedge (S_1, e_1) \xrightarrow{o_2} \ldots \xrightarrow{l_n} (s_n, e_n) \wedge (s_n, e_n) \xrightarrow{l_{n+1}} \ldots$$

taken according to rules (5.4). We say that a given observer $s' \in Obs_{\mathcal{S}}$ is reachable in $\mathcal{E}$ if there exists a generator $E' \in Gen_{\mathcal{E}}$ and an execution fragment from (s^0, E^0) containing (s', E').
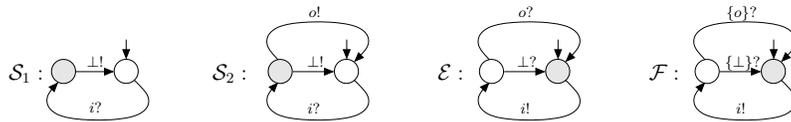
Figure 5.8: Counter example demonstrating that simulation does not preserve deadlock freeness. $In = \{i\}$, $Out = \{\perp, o\}$.

Definition 5.32. *A system $\mathcal{S}$ can deadlock in an environment $\mathcal{E}$, if there exists an observer $s \in Obs_\mathcal{S}$ and a generator $E \in Gen_\mathcal{E}$, such that (s, E) is reachable, and all labels on all executions of s in E only contain inputs and the empty output $\perp$.*

A system, which can deadlock in general, may not be able to deadlock in a specific environment, if its deadlock state is not reachable in this environment. Relativized simulation does not preserve deadlock freeness, either. Consider environment $\mathcal{E}$ on Fig. 5.8: $\mathcal{S}_1 \leqslant_\mathcal{E} \mathcal{S}_2$ and $\mathcal{S}_2$ is deadlock free in $\mathcal{E}$, while $\mathcal{S}_1$ can deadlock in $\mathcal{E}$.

Let us attempt to formalize deadlock freeness in the color-blind setting. A system $\mathcal{S}$ can *deadlock weakly* in a given color-blind environment $\mathcal{E}$, if execution in this environment can reach a state, such that all subsequent outputs will be observed by the environment in the same class as the empty output $\perp$. A system can *deadlock strongly* if it can reach a state, such that all subsequent outputs will be observed by the environment in the same class as the empty output $\perp$ and always infinitely many of those classes will be strictly smaller than *Out*.

Definition 5.33. *An execution of observer s_0 relative to a color-blind generator E_0 is a sequence of pairs of transitions:*

$$(s_0, E_0) \xrightarrow{i_1} (S_1, e_1) \wedge (S_1, e_1) \xrightarrow{(o_2, O_2)} \ldots \xrightarrow{l_n} (s_n, e_n) \wedge (s_n, e_n) \xrightarrow{l_{n+1}} \ldots ,$$

where $o_k \in O_k$ for even values of k. The transitions are taken according to the rules of (5.8). We say that a given observer $s' \in Obs_\mathcal{S}$ is reachable in $\mathcal{E}$ if there exists a generator $E' \in Gen_\mathcal{E}$ and an execution from (s^0, E^0) containing (s', E').

Definition 5.34. *A system $\mathcal{S}$ can weakly deadlock in a color-blind environment $\mathcal{E}$, if there exists an observer $s \in Obs_\mathcal{S}$ and a color-blind generator $E \in Gen_\mathcal{E}$, such that (s, E) is reachable, and all labels on all executions of s in E only contain inputs or output pairs (o, O) such that $\perp \in O$.*

Definition 5.35. *A system $\mathcal{S}$ can deadlock strongly in $\mathcal{E}$, if there exists an observer $s \in Obs_\mathcal{S}$ and a color-blind generator $E \in Gen_\mathcal{E}$, such that (s, E) is reachable, and all labels on all executions of $s \in E$ only contain inputs or output pairs (o, O) such that $\perp \in O$ and at least one of these executions contains infintely many labels where $O \neq Out$.*

We distinguish respectively weakly deadlock-free and strongly deadlock-free systems. Strongly deadlock-free systems are of greater interest, as due to our ubiquitous incorporation of the blind environment, all systems can weakly deadlock in almost every interesting environment.

Color-blind relativized simulation does not preserve deadlock freeness either. System $\mathcal{S}_2$ of Fig. 5.8 is deadlock-free in $\mathcal{F}$ (both weakly and strongly), whereas $\mathcal{S}_1$ deadlocks in $\mathcal{F}$ (both weakly and strongly). At the same time $\mathcal{S}_1 \leqslant_\mathcal{F} \mathcal{S}_2$.

Corollary 5.36. *None of the definitions of simulation presented in this chapter preserve deadlock freeness.*

A common and simple equivalence relation on systems is two-way simulation: $\mathcal{S}_1 \lessgtr \mathcal{S}_2 \iff \mathcal{S}_1 \leqslant \mathcal{S}_2 \wedge \mathcal{S}_2 \leqslant \mathcal{S}_1$. It is a standard result that two-way simulation does not preserve deadlock freeness.

Proposition 5.37. *None of the two-way simulations (usual,color-blind, relativized) based on simulations presented in this chapter preserve deadlock freeness.*

Proof. Fig. 5.9 shows the counter example: $\mathcal{S}_1 \lessgtr \mathcal{S}_2$, but $\mathcal{S}_2$ is deadlock free, while $\mathcal{S}_1$ can deadlock. Environments $\mathcal{E}$ and $\mathcal{F}$ form parts of a counter example for the relativized and relativized color-blind versions respectively. □

Not very surprisingly (relativized) bisimulation preserves deadlock freeness as we had defined it. As the result is fairly standard we only state it in the relativized form:

Proposition 5.38. *Let $\mathcal{S}_1, \mathcal{S}_2$ be two IOATSs and $\mathcal{E}$ be a color-blind IOATSs, such that $\mathcal{S}_1 \sim_{\mathcal{E}} \mathcal{S}_2$. If $\mathcal{S}_2$ is weakly deadlock free relative to $\mathcal{E}$, then also $\mathcal{S}_1$ is weakly deadlock-free relative to $\mathcal{E}$. If $\mathcal{S}_2$ is strongly deadlock-free relative to $\mathcal{E}$ then so is $\mathcal{S}_1$.*

Proof. Let us consider strong deadlock freeness first. Assume that $\mathcal{S}_1 \sim_{SS} \mathcal{S}_2$ and $\mathcal{S}_2$ is deadlock free, while $\mathcal{S}_1$ is not. This means that there exists a reachable pair $(s_1, E) \in Obs_1 \times Gen_{\mathcal{E}}$ such that all executions of s_1 in E only produce outputs observed in the same class as $\perp$ and at least one of these paths contains infinitely many observation classes strongly smaller than *Out*. But since $\mathcal{S}_1 \sim_{SS} \mathcal{S}_2$ there must be a reachable state $s_2 \in Obs_2$, such that the pair (s_2, E) is reachable, and s_2 has to be able to match all outputs of s_1 in the same observation classes. So it nearly qualifies as a strongly deadlocking system. The question is whether there exist an execution of s_2 which would not deadlock (produce something visible in other than the blind class of E). The answer is no. If it existed, s_1 would have to be able to match it and thus would not be a deadlocking state. As either s_1 is not deadlocking or s_2 is deadlocking, our initial assumption was incorrect, which shows that relativized bisimulation in a color-blind context preserves strong deadlock freeness.
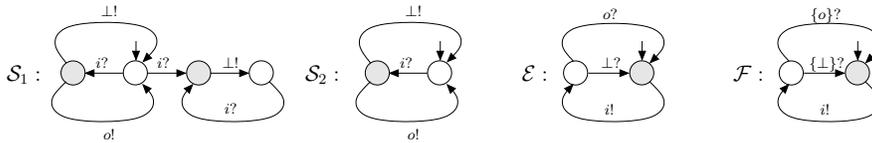
Figure 5.9: Counter example demonstrating that two way simulation does not preserve deadlock freeness. $In = \{i\}$, $Out = \{\perp, o\}$.

The result for weak deadlock freeness follows a very similar reasoning (just drop reservations about existence of the strong deadlocking path).   □

It is crucial that we can use a color-blind equivalence relation with its nice properties in order to specify refinement of systems. In future we want to provide algorithms, which decrease the size of the models (in the spirit of refinement). The correctness of these algorithms, shall rely on relativized bisimulation though, so they do not introduce deadlocks into systems.

5.5   Toward Engineering Design Languages

Until now we have proceeded under a strict assumption, namely that inputs and outputs of systems are atomic. This assumption does not hold for any, but the very simplest, of the engineering design languages. Realistic languages describe systems producing more structured outputs: sets, multisets, sequences or whole sequences of sets of atomic entities in a single reaction. We shall study two kinds of these advanced systems.

Assume a finite set of environment events *Event* and a finite set of atomic output actions *Action*. In order to be able to handle realistic languages we need to instantiate our framework for a given reaction style. This includes not only giving mappings from *Event* and *Action* to *In* and *Out*, but also proposing a suitable representation for observation classes and computing bounds on classes.

5.5.1   Set based

Let us begin with considering a variant of statecharts producing sets (see table 2.1, first row). The set of abstract events $In = Event$, while the set of abstract outputs contains all possible subsets of *Action*. The empty set is the empty output:

$$In = Event \qquad Out = \mathcal{P}(Action) \qquad \perp = \emptyset$$

Each global state $s \in \Sigma \times Store \times History \times Queue$ corresponds to a single observer at the abstract level, while a new generator is added for each pair of configurations (in practice it suffices to consider pairs of configurations related in the global transition relation):

$$Obs = \Sigma \times Store \times History \times Queue$$
$$Gen = (\Sigma \times Store \times History \times Queue)^2 \qquad\qquad (5.21)$$

Finally for every global transition $s \xrightarrow{e \quad o} s'$ at the model level (see rule 2.66, page 32) we introduce a single observation transition $s \xrightarrow{e?} (s, s')$ and a single

generation transition $(s, s') \overset{o!}{\longrightarrow} s'$ in the abstract IOATS. Remaining states are not related.

These definitions allow us to use the framework of previous sections and model environments for set producing statecharts as color-blind IOATSs. Note though, that, since *Out* is a powerset now, observation classes are not simply sets of outputs, but sets of subsets of *Action*. How should these classes be specified and represented? How can we compute the greatest lower bounds (glbs) and least upper bounds (lubs) on partitionings in this domain to efficiently obtain sums and products of IOATSs?

Subsets of finite sets, such as *Action*, are conveniently described with propositional formulæ. For each formula ϕ over variables representing elements of *Action* consider a corresponding set of its satisfiable assignments. Each assignment describes a set of actions, or a single output. We can use propositional formulæ instead of explicit enumerations as specifications of observational classes. More importantly we can efficiently implement them symbolically using BDDs.

We still need to make sure that classes represented on transitions leaving from a single observer are indeed disjoint and form a partitioning (see (5.9)). To achieve this we require that all corresponding formulæ are mutually exclusive and that they add up to the complete universum. For a set of formulæ $\phi_1, \ldots, \phi_n$ labeling all distinct transitions outgoing from a single observer state the following two conditions must hold:

$$\forall i, j \in \{1..n\}.\ i \neq j \ \Rightarrow \ \phi_i \wedge \phi_j \equiv \textit{false} \tag{5.22}$$

$$\phi_1 \vee \ldots \vee \phi_n \equiv \textit{true} \tag{5.23}$$

These conditions are feasible to verify computationally, especially easily using a BDD engine, or a SAT-solver.

Syntactically correct environments can be combined in sums and products using the operational rules presented in section 5.3. In particular the rules for observers rely on the existence of glbs and lubs for partitionings. These glbs and lubs exists (since all partitionings form a complete lattice under inclusion). We still need to give an efficient way to compute them:

Proposition 5.39. *Consider two equivalence relations $\sim_\phi$ and $\sim_\psi$ defined on $\mathcal{P}(\textit{Action})$, such that the observation classes of $\sim_\phi$ are described by formulæ $\phi_1, \ldots, \phi_m$ and observation classes of $\sim_\psi$ are described by formulæ $\psi_1, \ldots, \psi_n$. Then the equivalence relation $\sim_\phi \sqcap \sim_\psi$ is characterized by formulæ: $\{\phi_i \wedge \psi_j | i = 1 \ldots m,\ j = 1 \ldots n\}$.*

Obviously some of the new observational classes may be empty, since usually not all conjunctions are satisfiable. Unsatisfiable formulæ can be eliminated since corresponding BDDs automatically reduce to *false*.

The lub of two equivalence relations is a transitive closure of the union of these two relations. The computation of this transitive closure is realized by
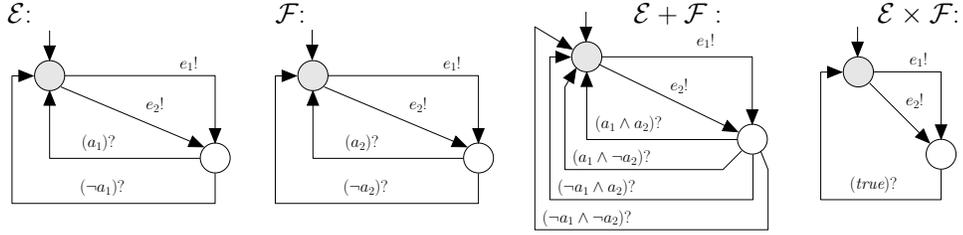
Figure 5.10: Left: environments $\mathcal{E}$ and $\mathcal{F}$, suitable for executing set-based reactive systems. Right: the sum and product of $\mathcal{E}$ and $\mathcal{F}$. Observational classes computed according to propositions 5.39 (for sum) and 5.40 (for product)

the classic Union-Find algorithm (see [66, section 2.3.3] or [21, chapter 21]) applied to the observation classes of both relations. Any two overlapping classes should be merged until no more classes overlap. An overlapping occurs if the conjunction of the two respective formulæ is satisfiable. A union of the class represented by ϕ with a class represented by ψ corresponds to replacement of the two formulæ with a disjunction $\phi \vee \psi$ of the two.

Proposition 5.40. *Let $\sim_\phi$, $\sim_\psi$ be equivalence relations on $\mathcal{P}(Action)$, such that their observation classes are described by formulæ $\phi_1, \ldots, \phi_m$ and $\psi_1, \ldots, \psi_n$ respectively. Then the equivalence relation $\sim_\phi \sqcup \sim_\psi$ is characterized by formulæ computed using the Union-Find algorithm applied to the set $\{\phi_1, \ldots, \phi_m, \psi_1, \ldots, \psi_n\}$, where two formulæ are unifiable, if their conjunction is satisfiable, and disjunction is used as the union operation.*

Figure 5.10 presents examples of environments with observational classes represented by propositional formulæ together with their sum and product, computed using the intersection and the union-find algorithm.

We remark, that nearly identical adaptation allows applications of our framework to other set-based languages including many hardware description languages and synchronous languages [11].

5.5.2 Sequence based

Several development languages for reactive synchronous systems reveal another output structure than sets. These languages, including Java Card [122] and variants of hierarchical statecharts (like UML [98] state diagrams), describe systems whose single reaction consists of a sequence of actions, instead of an unordered set.

$$In = Event \quad Out = Action^* \quad \bot = \langle\rangle \tag{5.24}$$

How should color-blind environments be specified for such systems? How can we compute the glbs and lubs on observational partitionings found in the respective kind of color-blind IOATS?

Similarly as in the set-based example, states of the actual system should be related with generators and observers at the abstract level and the transition relation of the modeling language with the observation and generation relations of our IOATS. The observation classes in the new IOATSs become sets of sequences of actions. If two sequences are in the same class, then they are considered equivalent by the environment. If we restrict ourselves to classes that are regular languages, then we can use a classifier DFA to describe the observation relation at each observer. A classifier DFA is a deterministic finite automaton, which instead of accepting strings, classifies them in groups.

Definition 5.41. *A classifier DFA over alphabet A is a quadruple $c = (S, A, s, \rightarrow)$, where S is a finite set of states, A is a finite set of symbols, $s \in S$ is an initial state and $(\rightarrow) \in S \rightarrow A \rightarrow S$ is an input-enabled transition function, meaning that for every state $s \in S$ it holds that $dom(\rightarrow(s)) = A$ (so $\rightarrow(s)$ is a total function). We write $s \xrightarrow{a} s'$, instead of $\rightarrow(s)(a) = s'$.*

A classifier DFA interprets a sequence of alphabet inputs by consecutively applying $\rightarrow$ to a state and the head of the input sequence obtaining a new state and input sequence. An execution over a list of symbols is abbreviated with $s \xrightarrow{\sigma} {}^* s_n$ meaning $s \xrightarrow{\sigma_1} s_1 \xrightarrow{\sigma_2} \ldots \xrightarrow{\sigma_{n-1}} s_{n-1} \xrightarrow{\sigma_n} s_n$, where σ_i is the ith symbol in the sequence σ.

Definition 5.42. *If $c = (S, A, s, \rightarrow)$ is a classifier DFA then we say that two sequences $\sigma_1, \sigma_2 \in A^*$ are equivalent with respect to c if their executions arrive at the same state: $\exists s'. \ s \xrightarrow{\sigma_1} {}^* s'$ and $s \xrightarrow{\sigma_2} {}^* s'$.*

Note that equivalence with respect to a classifier is indeed an equivalence relation and as such partitions the entire set of sequences A^* in a finite number of classes. The set of classes is isomorphic with the set of reachable states of the classifier. We shall use classifier DFAs to represent
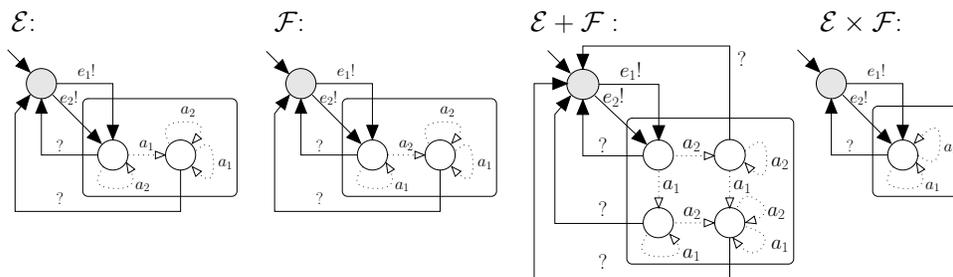
Figure 5.11: Two color-blind environments compatible with systems producing sequences of actions, their sum and product, computed with the algorithms of section 5.5.2.

the observation relation of color-blind environments observing sequences of outputs.

For each classifier $e = (S_e, Action, s_e, \rightarrow_e)$ consider $\gamma_e : S_e \rightarrow Gen$, a total mapping of classifier's states to generators. Each observer of the environment is a pair consisting of a classifier and such a generator mapping. A sequence of actions σ is first interpreted by e and then the generator mapping γ_e translates the resulting class to the next generator state. Formally:

$$(e, \gamma_e) \xrightarrow{\{\sigma \mid s_e \xrightarrow{\sigma}{}^* s\} ?} \gamma_e(s) \ \ .$$

Fig. 5.11 presents two color-blind IOATS compatible with systems of the following signature: $Event = \{e_1, e_2\}$, $Action = \{a_1, a_2\}$. Environment $\mathcal{E}$ distinguishes reactions containing at least one occurrence of a_1 from those sequences not containing a_1 at all. Similarly $\mathcal{F}$ distinguishes between sequences containing at least one a_2 from those not containing a_2 at all. Observers are drawn as boxes containing classifier DFAs. Classifier transitions are represented as dotted arrows to distinguish them from IOATS transitions.

Mechanical composition of such environments requires efficient ways of computing lubs and glbs on partitionings represented as classifiers. Both algorithms rely on the product construction:

Definition 5.43. *Let $e = (S_e, A, s_e, \rightarrow_e)$ and $f = (S_f, A, s_f, \rightarrow_f)$ be classifiers. A product of e and f is a classifier $e \otimes f = (S_e \times S_f, A, (s_e, s_f), \rightarrow)$, where $(s_e, s_f) \xrightarrow{a} (s'_e, s'_f)$ if $s_e \xrightarrow{a} s'_e$ and $s_f \xrightarrow{a} s'_f$.*

In practice unreachable states can be removed from the product.

The following proposition shows how computation of product of observers in rule (PO), section 5.3, is reduced to computing products of classifiers in this case:

Proposition 5.44. *Let $\sim_e$ and $\sim_f$ be two equivalence relations on $Action^*$ induced by classifiers e and f respectively. The greatest lower bound of the two relations $\sim_e \sqcap \sim_f$ exists and is induced by the product $e \otimes f$.*

Consider the example of Fig. 5.11 again. The sum of the two environments, $\mathcal{E} + \mathcal{F}$, is computed according to operational rules of section 5.3 (SG,PO) and the algorithm for glb of relations presented above. In particular the sum gives raise to an observer that distinguishes 4 classes of sequences: empty sequence, sequences consisting only of occurrences a_1, sequences containing only occurrences of a_2 and sequences containing occurrences of both a_1 and a_2.

The least upper bound of two partitionings $\sim_e \sqcup \sim_f$ is usually computed using a Union-Find algorithm, which unifies any two overlapping classes, until all classes are disjoint. In our case classes are represented by states in

the classifiers e and f. We need to apply the algorithm to states of e and f, ultimately producing a classifier, whose states are sets of states of f and e. The two classes s_1 and s_2 overlap, whenever there is an output sequence, that can advance one classifier to a state in s_1, and the other classifier to a state in s_2. The initial set of classes is given by reachable states of the product classifier $e \otimes f$:

   i. $S := \{\{e_i, f_j\} \mid (e_i, f_j)$ is reachable in $e \otimes f\}$.

   ii. If there exist $s_1, s_2 \in S$ such that $s_1 \cap s_2 \neq \emptyset$ then $S := (S \setminus \{s_1, s_2\}) \cup \{s_1 \cup s_2\}$.

   iii. Repeat (ii) until no more classes can be unified.

The final value of S is the set of states of the new classifier DFA. The initial state is the class that contains initial states of e and f (note that both of them will be in the same class). The transition function $\rightarrow$ is a sum of transition functions $\rightarrow_e$ and $\rightarrow_f$ lifted to sets of states. For $s_1, s_2 \in S$: $s_1 \xrightarrow{a} s_2$ if $\exists.p \in s_1.\exists p' \in s_2.p \xrightarrow{a}_e p' \vee p \xrightarrow{a}_f p'$. The following proposition claims that this function is well-defined, deterministic, and input-enabled:

Proposition 5.45. *Let $s_1, s_2 \in S$ be any two of the sets of states (not necessarily distinct) constructed with the above algorithm. Then for any states $p_1, p_2 \in s_1$, $p_1', p_2' \in s_2$ of the original classifiers and any symbol a: $p_1 \xrightarrow{a}_1 p_1'$ and $p_1' \in s_2$ iff $p_2 \xrightarrow{a}_2 p_2'$ and $p_2' \in s_2$, where $\longrightarrow_i$ denotes $\longrightarrow_e$ if $s_i \in S_e$ or $\longrightarrow_f$ if $s_i \in S_f$.*

It follows that the classifier $g = (S, A, s, \longrightarrow)$ constructed above is a well defined classifier DFA. Moreover, the observation classes that it induces are coarser than any class of $\sim_e$ and $\sim_f$. Due to the properties of the UNION-FIND algorithm, $\sim_g$ is actually the least equivalence encompassing both $\sim_e$ and $\sim_f$:

Proposition 5.46. *Let $\sim_e$ and $\sim_f$ be equivalences over Action*, induced by classifiers $e = (S_e, Action, s_e, \rightarrow_e)$ and $f = (S_f, Action, s_f, \rightarrow_f)$. The equivalence $\sim_e \sqcup \sim_f$ is induced by a classifier g such that its states are computed applying the UNION-FIND algorithm to the set*

$$\{ \{e_i, f_j\} \mid (e_i, f_j) \text{ reachable in } e \otimes f \} \ ,$$

where two sets s_1, s_2 are unifiable if $s_1 \cap s_2$ is not empty. The union operation is a set union, the initial state is the set containing initial states of e and f, and the transition function is a sum of transition functions lifted to sets of states.

The rightmost IOATS on Fig. 5.11 is a product of $\mathcal{E}$ and $\mathcal{F}$ obtained by application of the composition rules from section 5.3 (PG,SO) and the above algorithm for least upper bound of two equivalence relations on sets of sequences. The product gives rise to the observer which does not distinguish any sequences.

5.6 Example: Output Structure

In chapter 2 (see in particular 2.2.3) we have defined semantics for statecharts with various output structures. In section 5.5 we have seen how color-blind environments can be constructed for systems expressed in such languages. Now, we would like to provide an alternative formulation of earlier semantics with various output structures, without explicit parameterization. Instead of parameterizing the semantics with the constructors for outputs, as it was done in section 2.2.3, we accept one semantics, which only produces sequences of outputs, and then embed the system in various environments, to achieve various levels of sensitivity.

A system producing sequences can be turned into a set generating one by embedding it into an environment $\mathcal{SET}$ consisting of the following states and transitions:

$$\forall e \in \textit{Event}. \ \textbf{Set} \xrightarrow{i!} \textbf{set} \tag{5.25}$$

$$\forall p \in \mathcal{P}(\textit{Action}). \ \textbf{set} \xrightarrow{\{\sigma \ \mid \ elems(\sigma)=p\}?} \textbf{Set} \tag{5.26}$$

Now whatever analysis/transformation is to be done on $\mathcal{S}$ in some environment $\mathcal{E}$, it can be done on the environment $\mathcal{E} \times \mathcal{SET}$, which is equivalent to running the same analysis in the set-based setting.

Similarly if a bag b is a function $\textit{Action} \to \mathbb{N}$, then the environment $\mathcal{M}$-$\mathcal{SET}$ consisting of the following states and transitions:

$$\forall e \in \textit{Event}. \ \textbf{M-set} \xrightarrow{i!} \textbf{m-set}$$

$$\forall b \in \mathcal{M}(\textit{Action}). \ \textbf{m-set} \xrightarrow{\{\sigma \ \mid \ \forall a \in Action.b(a)=|\sigma \restriction \{a\}| \ \}?} \textbf{M-set}$$

can be used to turn a sequence based semantics into a multi-set based one.

Needless to say the perfect vision environment $\mathcal{V}$ preserves the sequence based semantics in such a context (and obviously being a 1 in the boolean algebra of environments it can be dropped altogether).

Construction of the environment **Inter** ignoring admissible interleavings in the output sequences (see row 4 in table 2.1) is slightly more complex. It requires knowledge about the structure of the model to be embedded in the environment or in structure of the outputs themselves. The latter approach has the advantage of keeping the environment model independent. Trees can be used as a precise semantics of the interleaving—use operator $\parallel$ (section 2.2.3). Once the outputs are seen as trees, each observation class has to identify all possible sequentializations (topological sortings) of some tree. We spare the technical details here.

5.7 Discussion of Discrimination

Definition 5.20 introduced a discrimination ordering on color-blind environments, using relativized simulation as the means for distinguishing systems. It is a valid question to ask whether a different criterion would not be more suitable, namely a two-way relativized simulation or the relativized bisimulation. In fact all these relations give raise to the same discrimination preorder, which we shall argue for now.

Definition 5.47. *Let $\mathcal{F}$ and $\mathcal{E}$ be two color-blind environments of the same signature: $\mathcal{E} \sqsubseteq \mathcal{F}$ iff $\forall \mathcal{S}_1, \mathcal{S}_2. \mathcal{S}_1 \lesssim_{\mathcal{F}} \mathcal{S}_2 \Rightarrow \mathcal{S}_1 \lesssim_{\mathcal{E}} \mathcal{S}_2$.*

Definition 5.48. *Let $\mathcal{F}$ and $\mathcal{E}$ be two color-blind environments of the same signature: $\mathcal{E} \sqsubseteq_{\sim} \mathcal{F}$ iff $\forall \mathcal{S}_1, \mathcal{S}_2. \mathcal{S}_1 \sim_{\mathcal{F}} \mathcal{S}_2 \Rightarrow \mathcal{S}_1 \sim_{\mathcal{E}} \mathcal{S}_2$.*

Theorem 5.49. *For any two environments with the same signatures:*

$$\mathcal{E} \sqsubseteq \mathcal{F} \iff \mathcal{E} \overline{\sqsubseteq} \mathcal{F} \iff \mathcal{E} \sqsubseteq_{\sim} \mathcal{F}$$

Proof. [case: $\mathcal{E} \sqsubseteq \mathcal{F} \Rightarrow \mathcal{E} \overline{\sqsubseteq} \mathcal{F}$]. Assume that $\mathcal{E} \sqsubseteq \mathcal{F}$. Take any two systems $\mathcal{S}_1$ and $\mathcal{S}_2$ such that $\mathcal{S}_1 \lessgtr_{\mathcal{F}} \mathcal{S}_2$. We need to show that $\mathcal{S}_1 \lessgtr_{\mathcal{E}} \mathcal{S}_2$. But $\mathcal{S}_1 \lessgtr_{\mathcal{F}} \mathcal{S}_2$ implies $\mathcal{S}_1 \leqslant_{\mathcal{F}} \mathcal{S}_2$, which together with $\mathcal{E} \sqsubseteq \mathcal{F}$ implies $\mathcal{S}_1 \leqslant_{\mathcal{E}} \mathcal{S}_2$. A symmetric argument leads to $\mathcal{S}_2 \leqslant_{\mathcal{E}} \mathcal{S}_1$, which conjoined lead to a conclusion that $\mathcal{S}_1 \lessgtr_{\mathcal{E}} \mathcal{S}_2$. So $\mathcal{E} \overline{\sqsubseteq} \mathcal{F}$.

[Case: $\mathcal{E} \overline{\sqsubseteq} \mathcal{F} \Rightarrow \mathcal{E} \sqsubseteq_{\sim} \mathcal{F}$]. We shall prove the contrapositive instead: $\mathcal{E} \not\sqsubseteq_{\sim} \mathcal{F} \Rightarrow \mathcal{E} \not\overline{\sqsubseteq} \mathcal{F}$. Assume $\mathcal{E} \not\sqsubseteq_{\sim} \mathcal{F}$, so there exist systems $\mathcal{S}_1, \mathcal{S}_2$ such that $\mathcal{S}_1 \sim_{\mathcal{F}} \mathcal{S}_2$ and $\mathcal{S}_1 \not\sim_{\mathcal{E}} \mathcal{S}_2$. We shall use $\mathcal{S}_1$ and $\mathcal{S}_2$ to construct new systems $\mathcal{S}_3$ and $\mathcal{S}_4$, such that $\mathcal{S}_3 \lessgtr_{\mathcal{F}} \mathcal{S}_4$ and $\mathcal{S}_3 \not\lessgtr_{\mathcal{F}} \mathcal{S}_4$.

Since $\mathcal{S}_1 \not\sim_{\mathcal{E}} \mathcal{S}_2$ there must exist a triple of states $(S_1, e, S_2) \in Gen_1 \times Obs_{\mathcal{E}} \times Gen_2$ and an observation class O_e such that both (S_1, e) and (S_2, e) are reachable and $e \xrightarrow{O_e?}\_$ and:

1° either there exists an output $o_1 \in O_e$ such that $S_1 \xrightarrow{o_1!}\_$ and for all o_2' such that $S_2 \xrightarrow{o_2'!}\_$, we have that $o_2' \notin O_e$,

2° or there exists an output $o_1 \in O_e$ such that $S_2 \xrightarrow{o_1!}\_$ and for all o_2' such that $S_1 \xrightarrow{o_2'!}\_$, we have $o_2' \notin O_e$.

Assume case 1°. At the same time environment $\mathcal{F}$, being both input and output enabled can execute $\mathcal{S}_1$ to a pair of states (S_1, f). Since $\mathcal{S}_1 \sim_{\mathcal{F}} \mathcal{S}_2$, $\mathcal{S}_2$ can match all actions of $\mathcal{S}_1$ on this execution path, reaching one of its states S_2'. In particular take O_f such that $o_1 \in O_f$ and $f \xrightarrow{O_f?}\_$. Then there exists $o_2 \in O_f$ such that $S_2' \xrightarrow{o_2!}\_$. So $o_1 \in O_e \cap O_f$, while $o_2' \in O_f \setminus O_e$.

Assume case 2°. Similarly environment $\mathcal{F}$ can execute $\mathcal{S}_2$ to a pair of states (S_2, f) and since $\mathcal{S}_1 \sim_{\mathcal{F}} \mathcal{S}_2$, $\mathcal{S}_1$ can match all actions of $\mathcal{S}_2$ on this execution path, reaching

one of its states S_1'. In particular take O_f such that $o_1 \in O_f$ and $f \xrightarrow{O_f?}$_. Then there exists $o_2 \in O_f$ such that $S_1' \xrightarrow{o_2!}$_. So $o_1 \in O_e \cap O_f$ and $o_2 \in O_f \setminus O_e$.

In both cases we have indicated two outputs, o_1 and o_2, such that $o_1 \in O_e \cap O_f$ and $o_2 \in O_f \setminus O_e$. Let $\mathcal{S}_3$ be an IOATS consisting of the execution of $\mathcal{S}_1$ until S_1, and a single generation transition $S_1 \xrightarrow{o_1!} x$. Observation transitions from all observers on the way, for all inputs not on this execution path, should also be directed to x. The observer x is a fixed arbitrary observer such that the main path is unreachable from it. The system $\mathcal{S}_4$ is identical to $\mathcal{S}_3$ only the transition generating o_1 is substituted by a transition generating o_2, though still targeting x. It is easy to see that $\mathcal{S}_3 \leqslant_{\mathcal{F}} \mathcal{S}_4$, since $\mathcal{F}$ is deterministic and the single path on which $\mathcal{S}_3$ and $\mathcal{S}_4$ differ always takes it to the same state, which cannot distinguish the difference. At the same time $\mathcal{S}_3 \nleqslant_{\mathcal{E}} \mathcal{S}_4$. $\mathcal{E}$ will always see the difference.

[case: $\mathcal{E} \sqsubset \mathcal{F} \Rightarrow \mathcal{E} \sqsubseteq \mathcal{F}$]. The proof of this implication is very similar to the previous one. We shall prove the contrapositive: $\mathcal{E} \not\sqsubseteq \mathcal{F} \Rightarrow \mathcal{E} \not\sqsubset \mathcal{F}$. Assume $\mathcal{E} \not\sqsubseteq \mathcal{F}$, so there exist systems $\mathcal{S}_1$, $\mathcal{S}_2$ such that $\mathcal{S}_1 \leqslant_{\mathcal{F}} \mathcal{S}_2$ and $\mathcal{S}_1 \nleqslant_{\mathcal{E}} \mathcal{S}_2$. We shall use $\mathcal{S}_1$ and $\mathcal{S}_2$ to construct new systems $\mathcal{S}_3$ and $\mathcal{S}_4$, such that $\mathcal{S}_3 \sim_{\mathcal{F}} \mathcal{S}_4$ and $\mathcal{S}_3 \nsim_{\mathcal{E}} \mathcal{S}_4$.

Since $\mathcal{S}_1 \nleqslant_{\mathcal{E}} \mathcal{S}_2$ there must exist a triple of states $(S_1, e, S_2) \in Gen_1 \times Obs_{\mathcal{E}} \times Gen_2$ and an observation class O_e such that both (S_1, e) and (S_2, e) are reachable and $e \xrightarrow{O_e?}$_ there exists an output $o_1 \in O_e$ such that $S_1 \xrightarrow{o_1!}$_ and for all o_2' such that $S_2 \xrightarrow{o_2'!}$_, we have that $o_2' \notin O_e$.

At the same time environment $\mathcal{F}$, being both input and output enabled can execute $\mathcal{S}_1$ to a pair of states (S_1, f). Since $\mathcal{S}_1 \sim_{\mathcal{F}} \mathcal{S}_2$, $\mathcal{S}_2$ can match all actions of $\mathcal{S}_1$ on this execution path, reaching one of its states S_2'. In particular take O_f such that $o_1 \in O_f$ and $f \xrightarrow{O_f?}$_. Then there exists $o_2 \in O_f$ such that $S_2' \xrightarrow{o_2!}$_. So $o_1 \in O_e \cap O_f$, while $o_2 \in O_f \setminus O_e$.

Let $\mathcal{S}_3$ be an IOATS consiting of the execution of $\mathcal{S}_1$ until S_1, and a single generation transition $S_1 \xrightarrow{o_1!} x$. Observation transitions from all observers on the way, for all inputs not on this execution path, should also be directed to x. The observer x is a fixed arbitrary observer such that the main path is unreachable from it. The system $\mathcal{S}_4$ is identical to $\mathcal{S}_3$ only the transition generating o_1 is substituted by a transition generating o_2, though still targeting x. It is easy to see that $\mathcal{S}_3 \sim_{\mathcal{F}} \mathcal{S}_4$, since $\mathcal{F}$ is deterministic and the single path on which $\mathcal{S}_3$ and $\mathcal{S}_4$ differ always takes it to the same state, which cannot distinguish the difference. At the same time $\mathcal{S}_3 \nsim_{\mathcal{E}} \mathcal{S}_4$. $\mathcal{E}$ will always see the difference.

$\square$

5.8 Beyond the Basics

Our setup only has focused on standalone/centralized control systems modeled in languages like statecharts. This was motivated by our initial requirements. It is a relevant question though how color-blindness could function in a distributed asynchronous setup. The first issue one encounters when

answering this question is that modeling distributed systems requires support for local hidden communication between components. It is traditional to model this kind of communications using hidden actions (τ-actions), but introduction of hidden actions to our framework invalidates the assumption about the environments being deterministic, and in consequence invalidates the proof of theorem 5.24.

Indeed it is now known that theorem 5.24 itself does not hold for non-deterministic environments, or more precisely it may be that $\mathcal{E} \sqsubseteq \mathcal{F}$, but not $\mathcal{E} \leqslant \mathcal{F}$. Consider the counter example on Fig. 5.12, where $In = \{i\}$ and $Out = \{o_1, o_2, o_3\}$. Environments $\mathcal{E}$ and $\mathcal{F}$ are distinguishing systems only by looking at their first output, and they become blind soon afterward. There are only nine pairs of systems differing in the first step, and it is easy to convince yourself that in all the nine cases the distinguishing abilities of $\mathcal{E}$ are identical with those of $\mathcal{F}$ (both can take apart each output from the others). So $\mathcal{E} \sqsubseteq \mathcal{F}$ and $\mathcal{F} \sqsubseteq \mathcal{E}$ but at the same time neither $\mathcal{E} \leqslant \mathcal{F}$ nor $\mathcal{F} \leqslant \mathcal{E}$. The reason is that the simulation relation for color-blind environments (definition 5.21) is too strong for nondeterministic systems. It requires that both environments are much more similar than needed for preserving discrimination properties.

A conjecture has been made about the form of the simulation suitable for nondeterministic color-blind IOATSs, but I have failed to prove theorem 5.24 for these formulations. It is known though, that the proof for general (nondeterministic) case may be very difficult. Kim Larsen describes this difficulty in his dissertation [69], where the proof takes 12 pages and is far from intuitive. We believe that that it is better to achieve more for deterministic environments in practical setting, than to struggle in achieving a more general theory, without the practical application, especially taking into account the fact that environments in practical applications, are indeed deterministic. Nevertheless nondeterministic environments remain a fascinating problem worth tackling in the future.

It seems that the setup for distributed systems should assume that all systems are modeled as IOATSs, and only the outermost environment of the entire network should be modeled as a color-blind IOATS. The crucial challenge lies in proposing an algorithm, which for a given system would infer an environment, based on the abilities of other connected systems and the most outer environment. This could possibly be done compositionally
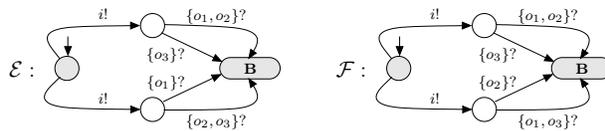
Figure 5.12: Counter example demonstrating mismatch of discrimination and simulation for non-deterministic color-blind IOATS.

by "peeling" layers out of the network: first compose the outer environment with all the systems that have a direct contact with it, creating a new most general color-blind environment for the next layer of the systems. Then the process should be iterated starting each time with the new outer environment and the inner part of the system. This highly appealing idea having its roots in [69] definitely deserves our consideration in the future.

Another theoretical challenge lies in generalizing theorem 5.49 into general blocking, asynchronous CCS with relativized simulation as in Larsen's dissertation.

A more practical direction would be to attempt incorporation of color-blindness into UML models (both behavioral and static). A good starting point seems to be phrasing environments as protocol state machines [103, pp.464-465] of UML, encoding color-blindness or extending UML with it.

5.9   Related Work

In [131] a static framework for specifying environments for reactive models is presented, which relies solely on state independent properties. This chapter has provided a theoretical foundation for a product line management setup similar to the one of [131], but based on behavioral properties. The static version did not support color-blind observations. Since the behavioral framework subsumes the static one, we have decided not to include it in this dissertation. However it seems perfectly possible to reformulate the theory of [131] in the fashion used here, explicitly indicating (stateless) environments, their discrimination properties and respective orderings. Then it shall easily follow that the behavioral framework is as expressive as the static one. Moreover model transformations that can be proved correct with respect to static assumptions, should also be correct with respect to their behavioral counterparts.

The material on color-blindness presented above has been recently accepted for publication [72]. The main difference introduced since [72] was written is the reformulation of the theory around relativized bisimulation. Previously the theory was based on simulation (and two way simulation) which under certain circumstances did not guarantee preservation of deadlock freeness (see section 5.4). The entire discussion of deadlock freeness is new in the present edition.

The theory of algebraic specification of concurrent processes (CCS) is due to Milner [94]. Relativized simulation has been originally introduced by Larsen in 1996, in his dissertation [70, 69, 73] written under Milner's supervision. Our framework is modeled after this work, rephrased in the setting of I/O alternating transition systems and extended with the notion of color-blindness. In Larsen's formulation, based on simple labeled transition systems [94], it was impossible to express environment's inability to

distinguish outputs.

The study of behaviors of systems embedded into execution contexts is relatively mature [69, 27, 84, 109, 58]. Our work stems out from this series, by its extended support for observability specifications via color-blindness. This support is needed, if the tools based on this framework, are to be useful for development of product lines of embedded systems. Prototypes of such tools, much in the spirit of Model Driven Architecture [99], are being developed. A paper [109] by Rajamani and Rehof is a good starting point for readers interested in deadlock-freeness preserving refinement—an approach alternative to the one we had in this chapter (using a deadlock-preserving equivalence in refinement-like applications).

5.10   Summary

We have presented a formalism for modeling environments of reactive synchronous systems. The essence of the formalism lies in the family of refinement and equivalence relations, which can be relativized and extended with color-blindness. The notion of color-blindness, which is new to our best knowledge, allows modeling dynamically changing abilities of the environment to observe mutations in output traces of systems. This in turn will be used later on in model transformations in our code-generator.

We have introduced the discrimination preorder, which was instrumental in defining composition operators. In chapter 6 we will see how this preorder supports incremental modeling of software product lines. We have proposed a simple operational characterization of this preorder by means of simulation. We have also discussed natural alternatives to our definition of discrimination and argued that they all coincide.

Last but not least we have demonstrated how the framework can be applied to realistic engineering design languages and argued that despite refinement oriented applications color-blindness allows use of equivalence to achieve similar purpose.

6

Product Line Derivation for Control Systems

In the previous chapter we have studied the theory of color-blind environments and their abilities to discriminate systems. We have hinted that these environments can be used as specifications for specialization of systems, and consequently for modeling software product lines. In this short chapter we shall pursue this idea further, by demonstrating a small example of a product line. The implementation of such a setup is a major task in itself, which we shall tackle separately in future projects. Nevertheless we will try to include as many hints as possible on the implementation details.

We broaden the meaning of a system to encompass an entire family of systems. Such a model (a statechart in our case) may represent functionality which in its entire richness may not be present in any of the actual systems being produced in the company. It may even contain contradicting behaviors, disambiguated by conditions on meta-states and meta-events (where meta-entities are usual states and events, meant to be eliminated during specialization). Particular systems are specified using models of environments. Each environment model depicts possible usage-observation scenarios for a given member.

We begin with a brief discussion of requirements for model transformations suitable in product line derivation (section 6.1). Then we present the example of alarm clock statechart (section 6.2) and models of environments specifying simplified alarm clocks (section 6.3). Section 6.4 discusses the challenges of implementation and other possible extensions. Section 6.5 is devoted to related work. We summarize the chapter in section 6.6.

The content of this chapter is one of the main outcomes of this thesis and the main motivation that led to development of the theory presented in chapter 5. I recommend reading at least sections 6.2–6.3 for the practical value they add to the content of chapter 5.

6.1   Requirements for Model Transformations

A tool supporting development of product lines would include a collection of model transformations, applied to the general family model. Each transformation can only be applied if its application precondition is satisfied. We face two proof obligations here. The first is a manual proof of correctness of the transformation itself, which can be done ahead of time. The second is the application precondition satisfaction check, which takes place at specialization time. This second proof should be obtained automatically using one of the available technologies (type checking, type inference, static analysis, model checking, or theorem proving).

Let T be a model transformation, $\mathcal{M}$ a family model and $\mathcal{E}$ an environment defining a specific family member. We say that T is correct iff the original model and the derived member are in $\mathcal{E}$-relativized bisimulation relation: $T(\mathcal{M}, \mathcal{E}) \sim_{\mathcal{E}} \mathcal{M}$. This means that the behavior of the two systems is virtually indistinguishable for $\mathcal{E}$.

A system is called strongly deterministic if its observation relation is deterministic, and there is only one outgoing edge from each generator (so that the response to the incoming input is entirely deterministic). Many engineering design languages and virtually all main programing languages are strongly deterministic. This fact can be used to reduce the correctness proof for transformations to one-way simulation proof.

Proposition 6.1. *Let $\mathcal{M}$ and $\mathcal{I}$ be strongly deterministic systems. Then for any compatible environment $\mathcal{E}$:*

$$\mathcal{I} \leqslant_{\mathcal{E}} \mathcal{M} \Rightarrow \mathcal{I} \sim_{\mathcal{E}} \mathcal{M} \ .$$

6.2   The Alarm Clock Example

Figure 6.1 depicts a statechart of an alarm clock $\mathcal{C}_0$ consisting of three state machines. The essential features of the alarm clock are handled by the timer machine. If the timer is in the armed state and the hardware sends an alarm time-out event, *alarmTO*, then the beeper is turned on. If the user wants to postpone the alarm he has to press the snooze button (event *snooze*), which allows him to continue sleeping until the snooze timer times out (event *snoozeTO*). The backlight machine controls whether the backlight in the alarm clock should be off, glowing or on. Only a faint light is displayed in the glowing state, such that the clock display can be read in the dark. The full light is on (state on) while the alarm is beeping or the snooze button is being pressed. The *snoozeR* event denotes the releasing of the snooze button. The sensor machine models a memory cell storing information about the level of light in the surroundings of the alarm clock. Proper events (*dark*, *bright*) are generated by the sensor driver whenever the ambient light passes above or below some threshold.
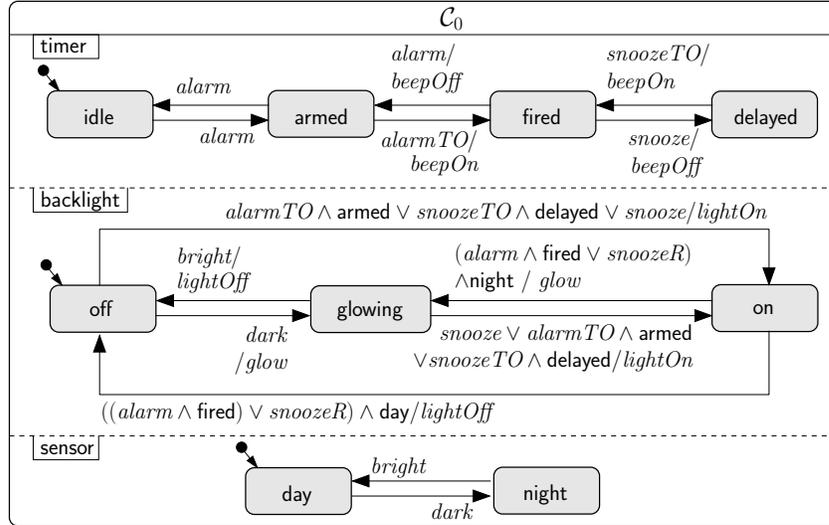
Figure 6.1: Model of a general alarm clock

We would like to support automatic derivation of variants for discrete control systems such as the this clock. One such variant C_1, which does not activate the backlight in reaction to the *snooze* button, is depicted on Fig. 6.2. Note the simplification of guards and two new transitions in the backlight state machine. What is the relation between the two models? Both models are indistinguishable for some execution environment, namely the one which becomes blind for the *lightOn* action immediately after producing the *snooze* event.

In a product line derivation scenario one usually assumes a domain model (the general clock of Fig. 6.1 in our example). Product line members are usually defined using specifications indicating how they should be constructed from the domain description. For control programs, we propose behavioral specifications in the form of execution environments.

6.3 Product Line of Alarm Clocks

We shall now present a family of environment specifications and a corresponding family of alarm clocks derived from the general alarm clock model, using the environments. In our presentations we shall use two simplifying shortcuts for expressing sets of observation classes (sets of labels of observation transitions). For a set of actions $A \subseteq \text{Action}$ let *ignore A* denote observation classes that ignore all elements of A, but distinguish all the other actions:

$$ \text{ignore } A = \left\{ \{o \cup o' | o' \in \mathcal{P}(A)\} \mid o \in \mathcal{P}(\text{Action} \setminus A) \right\} \qquad (6.1) $$
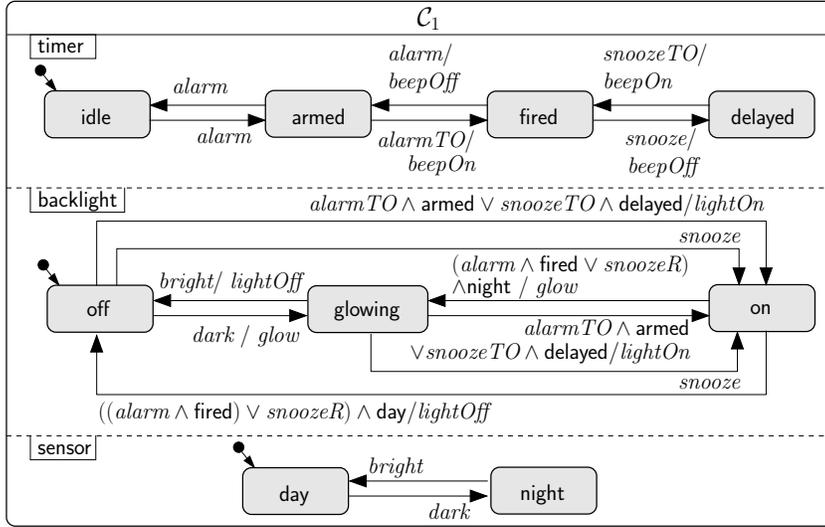
Figure 6.2: A specialized model, $\mathcal{C}_1$ of the alarm clock
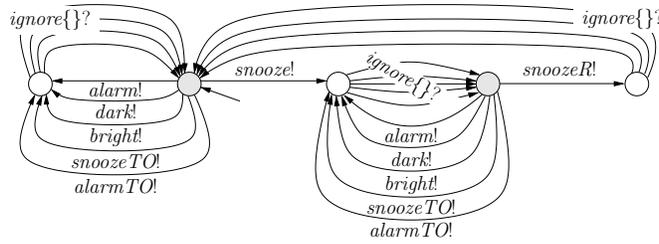
Figure 6.3: *Interleave snooze snoozeR*. Generation transitions to the blind observer **b** are not shown.

Note that ignoring the empty set, *ignore* {}, means observing any difference in outputs.

Another abbreviation *equiv A* denotes observation classes that are unable to distinguish between any actions in A:

$$equiv\ A = \big\{\{o \cup o' | o' \in \mathcal{P}(A) \setminus \emptyset\} \mid o \in \mathcal{P}(Action \setminus A)\big\}$$
$$\cup \big\{o \mid o \in \mathcal{P}(Action \setminus A)\big\} \tag{6.2}$$

In order to structure our specifications, we shall first state general requirements, which should hold for all the environments used to execute the alarm clock. These general requirements usually reflect the physical nature of actuators and sensors. In our case we state that *dark/bright* and *snooze/snoozeR* are always generated in an alternating fashion.

$$\mathcal{E}_0 = Interleave\ snooze\ snoozeR\ \wedge Interleave\ dark\ bright \tag{6.3}$$
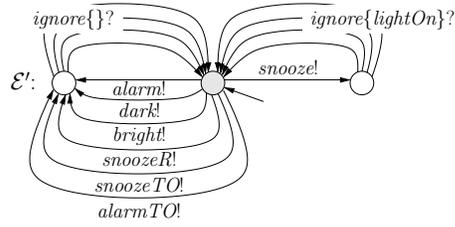
Figure 6.4: Environment $\mathcal{E}'$ ignoring the *lightOn* output produced in reaction to the *snooze* button.
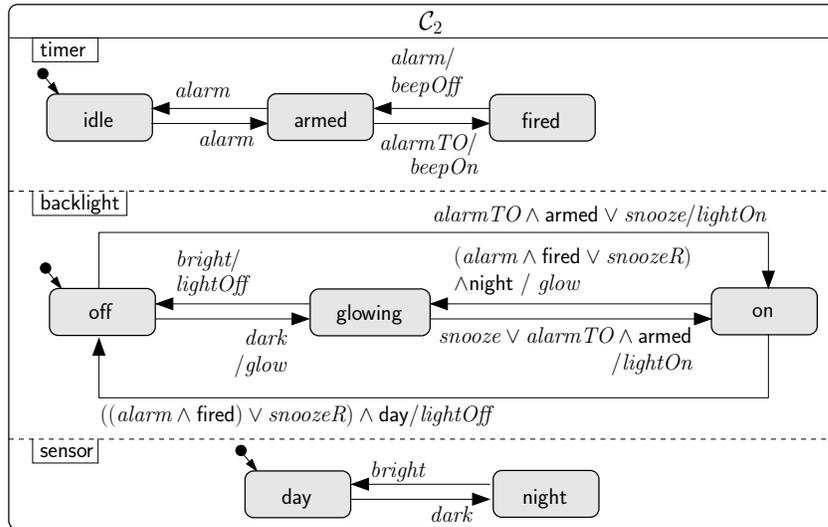
Figure 6.5: An alarm clock without the snooze function obtained by removing outputs/guards that are never observed/satisfied in $\mathcal{E}_2$, and then unifying states made indistinguishable (fired and snooze).

Figure 6.3 demonstrates how *Interleave* could be defined internally in a tool, using a set-based semantics.

The first member of the family, $\mathcal{C}_1$, was introduced in the previous section (Fig. 6.2). We have said that it operates in an environment which becomes blind for the *lightOn* action right after generating the *snooze* event. Formally $\mathcal{E}_1 = \mathcal{E}_0 \wedge \mathcal{E}'$, where $\mathcal{E}'$ is defined in Fig. 6.4.

Consider a new alarm clock variant $\mathcal{C}_2$, which is devoid of the actual snooze function (Fig. 6.5). The user of this clock can still press the snooze button, but the only effect it has is turning the backlight off for a short while. We can say that this user (environment) becomes blind to *beepOn* and *beepOff* actions initiated by the *snooze* and *snoozeTO* events. Formally $\mathcal{E}_2 = \mathcal{E}_0 \wedge \mathcal{E}''$, where $\mathcal{E}''$ is defined on Fig. 6.6.

The third clock variant $\mathcal{C}_3$ is a combination of $\mathcal{C}_1$ and $\mathcal{C}_2$. It has neither
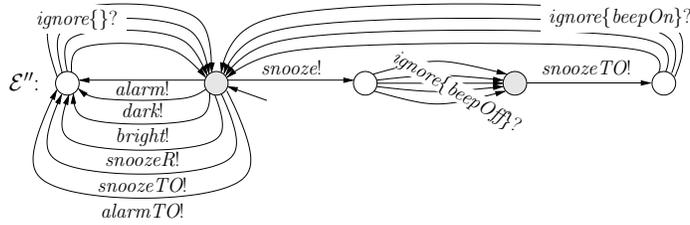
Figure 6.6: Environment $\mathcal{E}''$ ignoring the snooze function of the clock. Generation transitions to the blind observer **b** are not shown.
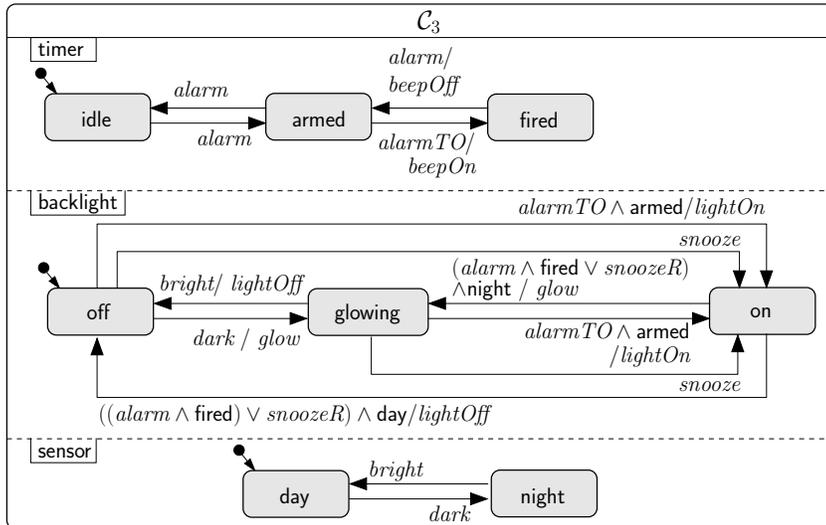
Figure 6.7: $\mathcal{C}_3$ combines limitations of $\mathcal{C}_1$ and $\mathcal{C}_2$
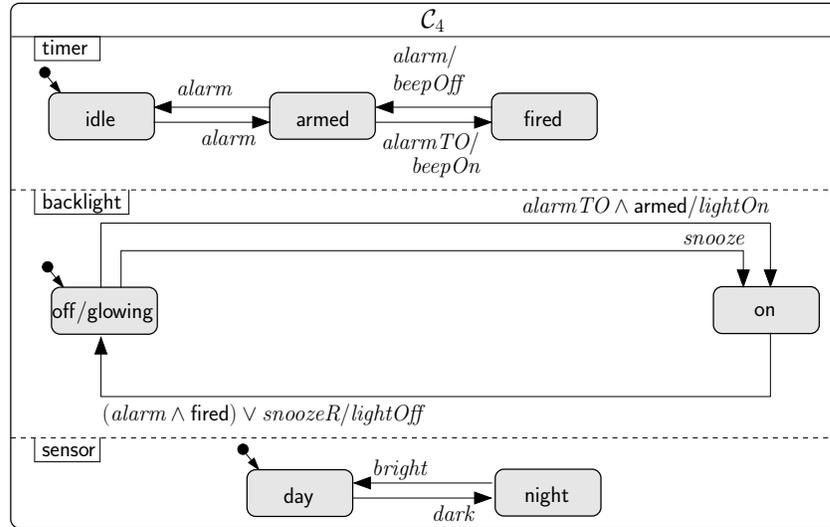
Figure 6.8: A model of the alarm clock with neither snooze related functions nor the glowing mode. The glow and off states are unified after showing that transitions connecting them have no side effects, transitions leaving them are identical and there are no guards distinguishing their activity.
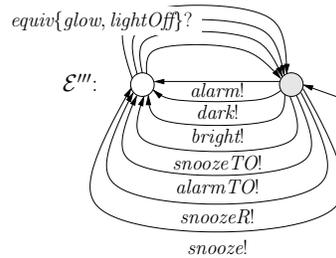
Figure 6.9: *Equiv glow lightOff*

the snooze function nor the snooze activated backlight function. We obtain it by specialization against the $\mathcal{E}_3$ environment, where $\mathcal{E}_3 = \mathcal{E}_1 \wedge \mathcal{E}_2$. The model is presented on Fig. 6.7. Note that this clock still needs a snooze button, which exhibits a slight anomaly in turning on the glow mode, namely that the glow mode will not be activated, while this button is pressed. This is a perfectly correct reminiscence of our original model, which could be easily remedied by adding another constraint to the environment, namely that event *snooze* never occurs.

We would like to consider yet another restriction of the clock behavior. The clock denoted $\mathcal{C}_4$ shall be deprived of the glowing mode (Fig. 6.8). The glow-mode lamp is not installed and the *glow* action is reimplemented to turn off the main lamp instead. A corresponding environment is $\mathcal{E}'''$ defined
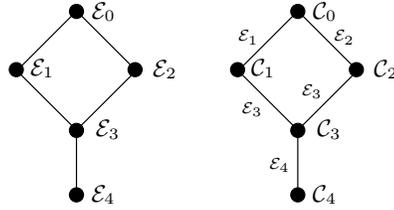
Figure 6.10: Left: simulation relationships between the environments. Right: relativized bisimulation relations between family members. Edges are labeled with environments parameterizing the relation.

on Fig. 6.9. This environment is itself interesting as it specifies a less shiny alarm clock, which may find its happy customers. Nevertheless, we decided to combine its characteristic with restrictions of $\mathcal{E}_3$, giving rise to an even simpler alarm clock with neither the snooze related functions nor the glow mode: $\mathcal{E}_4 = \mathcal{E}_3 \wedge \mathcal{E}'''$.

One can describe surprisingly many more reasonable variants even for such a simple system as our alarm clock. Even the minimal statechart model, which is a single state without any transitions, is a proper derivation from our alarm clock (with respect to the blind environment $\mathcal{B}$).

Figure 6.10 presents an overview of environments and systems in our product line. Edges represent simulation and relativized bisimulation. The following proposition explains how to interpret transitivity in the hierarchy of systems (the right side of the figure):

Proposition 6.2. *For any three systems $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_3$ and any two compatible color-blind environments $\mathcal{E}$ and $\mathcal{F}$ it holds that: $\mathcal{S}_1 \sim_\mathcal{E} \mathcal{S}_2 \wedge \mathcal{S}_2 \sim_\mathcal{F} \mathcal{S}_3 \wedge \mathcal{E} \leqslant \mathcal{F} \Rightarrow \mathcal{S}_1 \sim_\mathcal{E} \mathcal{S}_3$.*

6.4 Beyond the Basics

Even though the environments presented in the above example only consist of conjunctions of very few properties, they still becomes very large if one would like to expand them using rules 5.13–5.16. This has two consequences. First, complicated specifications should be created compositionally, as in the above example; preferably relying on a handful of predefined parameterized environment templates, as we did above. A convenient set of basic templates still needs to be found and this cannot be achieved without implementing the entire system and running several case studies on realistic examples. We foresee that in a product line tool the environments will not be drawn as state machines, but instead composed of simple textual expressions. The knowledge of predefined sets of available expressions can boost the automatic analysis technology, too.

Second, it seems infeasible to maintain environments in explicit state machine form during analysis, as they are likely to explode in size. Instead it is preferable to analyze the composition of the system and several environments at the same time. This is relatively easy to achieve if we only allow conjunctions in environment expressions. We have designed provisional algorithms achieving this and intend to publish them in near future.

Another important fragment of the road map to completely automatic derivation of family members is the right set of model transformations. We have devised a proposal of such a set and proved most of the transformations correct. The remaining obstacle is to implement the transformations (easy) and the proof machinery for verifying transformation preconditions (hard). Currently we work on a prototype of the latter, which is based on simple forward reachability analysis with BDDs. However, we do not expect overwhelming results with this method. Models like $\mathcal{C}_0$ can indeed be analyzed within seconds, but this method is hardly going to scale up to industrial size models (hundreds of transitions). The forward reachability method computes the complete reachable state space, before transformations can be applied. The preconditions of transformations are usually predicates over states, which can then be easily checked. The problem is that for larger models, reachability is not likely to terminate in reasonable time in the first place, thus not letting the specializer apply the transformations ever.

As an alternative we think of employing a mixture of program analysis and partial reachability mechanisms. Experience [82, 81] shows that it is often more efficient to make many separate small proofs instead of computing the whole reachable state space and reason based on it. The compositional and partial method of [81] works exactly with the same kind of systems we are dealing with. We hope that we can devise a similar set of techniques for dealing with proofs of transformation preconditions. The author is currently working on this problem within a new project in the Embedded Software Systems Center (CISS) at Aalborg University.

6.5   Related Work

Derivation of product lines can be implemented using partial evaluation technology [63, 24, 48]. There have been some limited attempts to enable partial evaluation based on execution traces instead of specific input values (see [54, 97, 34]), nevertheless these were never implemented for realistic engineering languages. We fear that these abstract process calculi transformations can be barely applied in such contexts. This is why we propose a bottom up approach, where transformations are defined on the actual language level, and proved correct on the abstract level. Our framework allows more transformations than known before due to the color-blindness, which allows some non-reductive mutations in the program.

As we have mentioned before, we originally proposed derivation of product lines for programs with explicit control flow in [131]. The behavioral theory that came in [72] and in chapter 5 has been mostly motivated by requirements set by [131] for the static setting. In that paper we postulated hierarchical (or lattice-like) shapes of product families. We demonstrated a simple product line of CD players, shaped similarly to our family of alarm clocks, from the present text. In [131] we have also proposed a specialization algorithm, which we have not yet extended for the behavioral version. This algorithm may be of interest for reasons of scalability. Although the older static framework offers less expressive product specifications, it is easier to implement efficiently.

6.6 Summary

This chapter was devoted to practical demonstration of application of theory introduced in chapter 5. We have indicated that generation of product family members can be carried out by means of model transformations which transform the model of the entire family under an assumption of specific execution environment. We have required that the result of such transformation is equivalent to the original model up to relativized bisimulation.

Then we have shown a statechart model of a simple alarm clock and a lattice of various execution environments, each presenting a different usage scenario of the alarm clock. We arrived at an isomorphic lattice of corresponding alarm clocks, related by relativized bisimulation. Finally we have sketched the main obstacles on the road to automatic derivation of such model families, which is the topic of our ongoing work.

7

Conclusion

Let me briefly summarize the achievements of this thesis. The departing point for our investigations was the formal definition of the statechart language, which was parameterized with various priority orderings and output structures. This allowed us to simultaneously cover multiple variants of statecharts in some of the later investigations, such as the color-blind theory. In particular it led us to the algorithm for static conflict resolution, which is independent of any particular conflict resolution strategy.

We have studied several classes of approaches to code generation, finding that the interpretative method (based on runtime model representation and an interpreter) is the most suitable for our requirements: both because of its high memory efficiency and due to good behavior with respect to fault tolerance in systems without memory protection. We have discussed the construction of a modular, retargetable code generator, SCOPE, based on the interpretative method. In order to simplify the runtime interpreter, we proposed two algorithms that replace interpreter logic with additional transitions in the runtime representation of the model: elimination of dynamic scopes and static conflict resolution. Similar algorithms can be proposed for other contrived elements of the statechart language not covered in this thesis (in particular do reactions, forks and joins).

We have deeply analyzed two code generation methods implemented in SCOPE: one preserving the hierarchy and one based on flattening. We have found that, even though the hierarchical code generator scales well and beats the industrial implementations, the flattening back-end performs even better in majority of cases. This was possible due to a novel flattening algorithm, which only increases the size of the model polynomially. The existence of such an algorithm was seriously doubted previously. We have shown that an algorithm, which does not exploit the internal asynchronous message passing mechanism (signals), is doomed to perform miserably on deep and highly concurrent models. Our flattening code generator beats the visualSTATE implementation by approximately 30% on the middle sized

industrial examples and by up to 80% on the artificially created examples. The code generated by SCOPE uses very little writable memory (as low as 15-30 bytes plus the size of the current state vector).

In the second part of the thesis we have explored theoretical topics related to specifications of correctness of code generation, and indirectly to modeling of software product lines for embedded systems. We have abstracted from the statechart language itself, by considering transition systems suitable for modeling all similar languages (maintaining input enabledness and synchronicity). We enriched the language of environments with the notion of color-blindness: a dynamic inability to observe variations in system outputs. We have formalized discrimination properties of environments and then proved an appealing characterization of the discrimination ordering by means of simulation. This ordering became a base for shaping product families in lattice like structures.

As we did for system models, we have also studied various output structures for environments. This is particularly important for environments, since they shall be used as specifications for optimizers. Simpler output structures (such as sets) are more automatically tractable than more expressive ones (sequences).

After having performed the steps necessary to instantiate the abstract framework for concrete languages like statecharts, we considered a simple statechart example and modeled a family of its variants by means of colorblind environments. We obtained a flexible, behavioral language for modeling families of discrete control programs, which supports stepwise hierarchical development.

In all this work on code generation I have examined and created numerous models of embedded systems, among others studying the works of alarm clocks, CD players, microwave ovens, light switches, thermostat controllers, vending machines, photo cameras and telephones. These models originated from the industrial partners (some of them anonymized or obfuscated), student projects, case studies, research papers and from my personal artificial creativity. To make sure that my investigations were realistic, I have inspected instruction sets and architectures of popular microcontrollers, primarily Atmel AVR/ATmega and Hitachi's H8/300, the compilers and libraries from the embedded world (kindly supplied by IAR Systems), and the source code of some real embedded applications developed in C and assembly languages.

I have translated several statecharts to programs manually and devised a high level environment for simulating and testing statecharts (an interpreter coded in Standard ML and a collection of test scripts). I have myself implemented several code generators (various back-ends of SCOPE and Charter) and supervised student projects related to code generation and UML.

Today I am strongly convinced that model driven development not only leverages the understanding of control algorithms used in the embedded

world, but also can integrate and boost various aspects of software engineering: ranging from traditional problems of design and implementation (translation in our case), via formal proofs of correctness (model checking), automatic testing, monitoring and product line derivation (specialization). I believe that it is possible to apply it even for embedded systems with very constrained resources. In particular I have shown this by proposing an efficient translation scheme for statecharts which beats current industrial implementations and by developing a framework that integrates discrete control models with product family modeling.

In the near future I expect to focus primarily on applications of color-blind environment models to automatic derivation of product lines, automatic testing and system monitoring (runtime verification).

Appendix A

Quantum Programming Example

The following listing shows a simplified version of Samek's [114] execution engine—an abstract class `QHsm` implementing a state machine. Concrete statecharts are implemented as a descendants of `QHsm` in this framework. States of the statechart as modeled as functions (or more precisely pointers to functions) handling events.

I have removed some optimizations and simplified the engine, so that it is more apparent what is going on. The original engine is bigger, handles more cases, may work faster occasionally. It also uses more space. Bear in mind that Samek is not observing UML semantics. Most importantly he does not handle concurrency, and whenever a transition is fired, he first executes its action, then he exits the scope (firing respective exit actions) and enters the target.

```
/* qhsm.cc: Samek's quantum engine for executing a rectricted
            version of statecharts.

   Miro Samek.  Practical Statecharts in C/C++. CMP Books. 2002
   Simplifications by Andrzej Wasowski, IT University of Copenhagen.
*/

#include <assert.h>
#define TRIGGER(state_, sig_)  ((QState)(this->*(state_))(sig_))

typedef int QEvent;
enum { Q_empty=0, Q_entry, Q_exit, Q_init, Q_user };

class QHsm {

public:
  typedef void (QHsm::*QPseudoState)(QEvent const);
  typedef QPseudoState (QHsm::*QState)(QEvent const);
```

```
    QHsm(QPseudoState initial) : myState(&QHsm::topstate),
             mySource((QHsm::QState)initial) {};

    virtual ~QHsm() {};

    void init(QEvent const e = 0) {
      QState s = myState;
      (this->*(QPseudoState)mySource)(e);
      s = myState;
      TRIGGER(s, Q_entry);

      while (TRIGGER(s, Q_init)==0) {
        s = myState;
        TRIGGER(s, Q_entry);
      }
    }

    void dispatch(const QEvent e) {
      mySource = myState;
      while (mySource)
        mySource = (QHsm::QState)(this->*mySource)(e);
    }


protected:
  QPseudoState topstate (const QEvent) { return 0; }  // the root
state
  void tran(QState target);
  void init_(QState target) { myState = target; }

private:
  QState mySource; // the source of currently taken transition
                          // needed for dynamic scope computation
  QState myState;

};

// fires a transition targeting 'target'
void QHsm::tran(QState target) {
  QState entry[8]; // 8 is assumed maximum depth of the hierarchy
  QState p, q, s, *e, *lca;

  for (s=myState; s!=mySource; ) {
    QState t = TRIGGER(s, Q_exit);
    if (t) s = t;
    else s = TRIGGER(s,Q_empty);
  }
```

```
e = entry;
e[0] = 0;
*(++e) = target; // assume entry to target

// (a) check mySource == target (transition to self)
if (mySource == target) {
  TRIGGER (mySource, Q_exit); // exit source
  goto inLCA;
}

// (b) check mySource == target->super
p = TRIGGER(target, Q_empty);
if (mySource == p) goto inLCA;

// (c) check mySource-->super == target->super (most common)
q = TRIGGER(mySource, Q_empty);
if (q == p) {
  TRIGGER(mySource, Q_exit);
  goto inLCA;
}

// (d) check mySource-super == target
if (q == target) {
  TRIGGER(mySource, Q_exit);
  --e;
  goto inLCA;
}

// (e) check rest of mySource == target->super->super hierarchy
*(++e) = p;
for (s = TRIGGER(p, Q_empty); s; s = TRIGGER(s, Q_empty)) {
  if (mySource == s) goto inLCA;
  *(++e) = s;
}
TRIGGER(mySource, Q_exit);

// (f) check rest of mySource->super == target->super->super->...
for (lca = e; *lca; --lca) {
  if (q == *lca) {
    e = lca - 1;
    goto inLCA;
  }
}

// (g) check each mySource->super->super ... for each target ...
for  (s = q; s; s = TRIGGER(s, Q_empty)) {
  for (lca = e; *lca; --lca) {
    if (s==*lca) {
      e = lca - 1;
```

```
        goto inLCA;
      }
    }
    TRIGGER (s, Q_exit);
  }
  assert(0); // should never reach here---malformed statechart

 inLCA:
  while (s = *e--) TRIGGER(s, Q_entry);
  myState = target;
  while (TRIGGER(target, Q_init) == 0) {
    target = myState;
    TRIGGER(target, Q_entry);
  }
}

typedef QHsm::QPseudoState QState;
```

A statechart of Fig. 3.1 is implemented below as a descendant of the `QHsm` class. It is supposed to implement the same functionality as the code of Fig. 3.2, modulo semantic variations by Samek (different order of actions when transitions are fired). The code excerpt finishes with a dummy driver for this statechart in QP framework. Again the driver replaces the code of Fig. 3.3, for QP based implementations.

```
/* 2005 (c) Andrzej Wasowski, IT University of Copenhagen
   qp.cc: Implementation of the statechart of figure
   images/state-pattern.fig using Samek's QP framework. */

#include "qhsm.cc"
#include "actions.cc"

enum { e3 = Q_user };

class Topstate : public QHsm {
public:
  Topstate() : QHsm((QHsm::QPseudoState)(&Topstate::initial)) {};
private:
  void initial(QEvent const);
  QState M(QEvent e);
      QState K(QEvent e);
      QState L(QEvent e);
  QState P(QEvent e);
};

QHsm::QState Topstate::M(QEvent e) {
  switch (e) {
    case Q_init:
      init_((QState)&Topstate::K);
```

```
      return 0;
    case Q_entry:
      o7();
      return 0;
    case Q_exit:
      o8();
      return 0;
    case e3:
      o14();
      tran((QState) &Topstate::P);
      return 0;
  };
  return (QHsm::QState)&Topstate::topstate; //inherited
};

QHsm::QState Topstate::K(QEvent e) {
  switch (e) {
    case Q_entry:
      o9();
      return 0;
    case Q_exit:
      o10();
      return 0;
    case e3:
      o13();
      tran((QState) &Topstate::L);
      return 0;
  };
  return (QHsm::QState)&Topstate::M;
};

QHsm::QState Topstate::L(QEvent e) {
  switch (e) {
    case Q_entry:
      o11();
      return 0;
    case Q_exit:
      o12();
      return 0;
  }
  return (QHsm::QState)&Topstate::M;
};

QHsm::QState Topstate::P(QEvent e) {
  switch (e) {
    case Q_entry:
      o5();
      return 0;
    case Q_exit:
```

```
      o6();
      return 0;
  }
  return (QHsm::QState)&Topstate::topstate;
};

void Topstate::initial(QEvent const e) {
  init_((QState)(&Topstate::M));
}

// a dummy driver:
int main (void) {
  Topstate  t = Topstate();
  t.init();
  while (1) t.dispatch (e3);
}
```

The above program compiled with g++ for x86, including the QHsm engine and dummy drivers produces an executable of 6400 bytes.

```
wasowski@klimt$ g++ -v
Reading specs from /usr/lib/gcc-lib/i686-pc-linux-gnu/3.3.5-20050130/specs
Configured with:
/var/tmp/portage/gcc-3.3.5.20050130-r1/work/gcc-3.3.5/configure
--enable-version-specific-runtime-libs --prefix=/usr
--bindir=/usr/i686-pc-linux-gnu/gcc-bin/3.3.5-20050130
--includedir=/usr/lib/gcc-lib/i686-pc-linux-gnu/3.3.5-20050130/include
--datadir=/usr/share/gcc-data/i686-pc-linux-gnu/3.3.5-20050130
--mandir=/usr/share/gcc-data/i686-pc-linux-gnu/3.3.5-20050130/man
--infodir=/usr/share/gcc-data/i686-pc-linux-gnu/3.3.5-20050130/info
--with-gxx-include-dir=/usr/lib/gcc-lib/i686-pc-linux-gnu/\
3.3.5-20050130/include/g++-v3
--host=i686-pc-linux-gnu --disable-altivec --enable-nls
--without-included-gettext --with-system-zlib --disable-checking
--disable-werror --disable-libunwind-exceptions --disable-multilib
--disable-libgcj --enable-languages=c,c++ --enable-shared
--enable-threads=posix --enable-__cxa_atexit --enable-clocale=gnu
Thread model: posix gcc version 3.3.5-20050130 (Gentoo Linux
3.3.5.20050130-r1, ssp-3.3.5.20050130-1, pie-8.7.7.1)

wasowski@klimt$ g++ -DNDEBUG -o qp -Os qp.cc -lstdc++ && strip qp
wasowski@klimt$ ls -l qp
-rwxr-xr-x  1 wasowski wasowski 6400 wrz  8 20:24 qp
```

Appendix B

SCOPE Hierarchical Engine

```
/* File : CodGenC1.c
 * Comment: Runtime interpreter for programs generated with CodGenC1
 *
 * Copyright (C) 2001-2004 Andrzej Wasowski
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation; either version 2 of
 * the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA
 * 02111-1307, USA.
 */

/**
 * a hack to compile an avr version easily (unfortunately no assert
 * in avr lib, at least in the one I got installed)
 */

#ifdef NDEBUG
#define assert(p)
#else
#include <assert.h>
#endif


#define SCOPE_VERBOSE 0
```

```
#if (SCOPE_VERBOSE && !defined(NDEBUG))
  #define VERBOSE(msg)     { printf ( "%s", msg ); }
  #define VERBOSE_OR(msg,s)  { printf( "%s", msg ); \
            print_or_sym(s); \
            printf( "\n" ); }
  #define VERBOSE_AND(msg,s)  { printf( "%s", msg ); \
            print_and_sym(s); \
            printf( "\n" ); }
#else
  #define VERBOSE(msg)
  #define VERBOSE_OR(msg,s)
  #define VERBOSE_AND(msg,s)
#endif

#ifdef SCOPE_CONFSET

/* next_conf size */
static confiter neco_size = 0;
/* prev_conf size */
static confiter prco_size = 0;

#endif

/* targets size */
static tgtsiter tgts_size = 0;

#ifndef NDEBUG

#include <stdlib.h>
#include <stdio.h>

/* check if targets are empty (used in assertions) */
static int targets_empty(void)
{
  return tgts_size == 0;
}

/* check if conf is empty (used in assertions) */
static int conf_empty(conf c)
{

#ifdef SCOPE_CONFSET
  if (c == next_conf)
    return (neco_size == 0);
  else
    return (prco_size == 0);
#else
  int i = 0;
```

```
  for (; i < STATE_WIDTH; i++)
    if (c[i] != STMRK)
      return 0;
  return 1;
#endif

}

/* check if state s is a member of c */
int conf_member_ext(conf c, andstref s)
{

#ifdef SCOPE_CONFSET
  confiter i = 0;
  confiter size = (c == next_conf) ? neco_size : prco_size;

  for (; i < size; ++i)
    if (c[i] == s)
      return 1;
  return 0;
#else
  return c[*(GET_AND(s))] == s;
#endif

}

void next_conf_reset(void)
{

#ifdef SCOPE_CONFSET
  neco_size = 0;
#else
  confiter i;

  for (i = 0; i < STATE_WIDTH; ++i)
    next_conf[i] = STMRK;
#endif

}

int compar(const void *a, const void *b)
{
  if (*(andstref *) a < *(andstref *) b)
    return -1;
  else if (*(andstref *) a == *(andstref *) b)
    return 0;

  return 1;
```

```
}

void print_and_sym(andstref s)
{
  int c = 0;

  while (1) {
    if (and_syms[c].id == s) {
      printf("%s ", and_syms[c].name);
      break;
    } else ++c;
  }
}

void print_or_sym(orstref s)
{
  int c = 0;
  while (1) {
    if (or_syms[c].id == s) {
      printf( "%s", or_syms[c].name);
      break;
    } else ++c;
  }
}

/* check if 'what' is an andstate. Only works in --debug mode,
 * as it uses symbolic dictionaries
 */
unsigned is_andstate(andstref what)
{
  int c = 0;

  while (and_syms[c].id != 0) {
    if (and_syms[c].id == what)
      return 1;
    c++;
  }
  return 0;
}

/*
 * check if 'what' is an orstate. Only works in --debug mode,
 * as it uses symbolic dictionaries
 */
unsigned is_orstate(orstref what)
{
  int c = 0;

  while (or_syms[c].id != 0) {
```

```
      if (or_syms[c].id == what)
        return 1;
      c++;
    }
    return 0;
}


void dump_conf()
{
    int i = 0;
    conf myconf;

#ifndef SCOPE_CONFSET
    int neco_size = STATE_WIDTH;
#endif
    for (i = 0; i < neco_size; ++i)
      myconf[i] = next_conf[i];

    printf("conf: [");
    qsort(myconf, neco_size, sizeof(andstref), compar);
    for (i = 0; i < neco_size; ++i)
      if (myconf[i] != STMRK)
        print_and_sym(myconf[i]);
    printf("]\n");
}

VS_BOOL activeand_next(const andstref what)
{
#ifdef SCOPE_CONFSET
    andstref min = *(GET_AND(what) + 1);
    confiter i = 0;

    assert(min > STMRK);

    /* you should never ask for sthg which is not a state */
    assert(what != STMRK);
    assert(what != HMRK);
    assert(what != IMRK);
    assert(is_andstate(what));
    /* you should never need to test activity of rootstate */
    assert(*GET_AND(what) != STMRK);
    /* or in other words (should be a redundant test): */
    assert(what != ROOT_STATE);
    /* non-state will never cause activity */
    assert(STMRK < what);

    for (; i < neco_size; ++i)
      if (next_conf[i] >= min && next_conf[i] <= what)
        return 1;
```

```
  return 0;
#else
  assert(what != ROOT_STATE);
  assert(*GET_AND(what) < STATE_WIDTH);
  return next_conf[*GET_AND(what)] == what;
#endif
}

VS_BOOL activeor_next(const orstref what)
{
#ifdef SCOPE_CONFSET
  /* inefficient but ok for testing */
  const horcell * i = GET_OR(what);
  do {
    if ( activeand_next(*i) ) return 1;
    i++;
  } while ( *(i-1) < *i );

  return 0;
#else
  assert( what < STATE_WIDTH );
  return next_conf[what] != STMRK;
#endif
}

#endif /* NDEBUG */

VS_BOOL activeand(const andstref what)
{
#ifdef SCOPE_CONFSET
  andstref min = *(GET_AND(what) + 1);
  confiter i = 0;

  assert(min > STMRK);

  /* you should never ask for sthg which is not a state */
  assert(what != STMRK);
  assert(what != HMRK);
  assert(what != IMRK);
  assert(is_andstate(what));
  /* you should never need to test activity of rootstate */
  assert(*GET_AND(what) != STMRK);
  /* or in other words (should be a redundant test): */
  assert(what != ROOT_STATE);
  /* non-state will never cause activity */
  assert(STMRK < what);

  for (; i < prco_size; ++i)
    if (prev_conf[i] >= min && prev_conf[i] <= what)
```

```
      return 1;
  return 0;
#else
  return prev_conf[*GET_AND(what)] == what;
#endif
}


#ifdef SCOPE_SIGNALQUEUE

/* Signal Queue - implemented as a round buffer */
static queueiter qbeg = 0;
static queueiter qend = 0;

/*
 * silently assume that queue is long enough! which shall be the
 * case, since verification returns this result
 */
void enqueue1(eventtag e)
{
  assert(e < EVENTNO);

  queue[qend] = e;

  if (qend == QUEUE_LENGTH - 1)
    qend = 0;
  else
    qend++;

  assert(qend != qbeg);
}


#ifdef SCOPE_SLIST
void enqueueList(signalref ref)
{
  while (signals[ref] != NOEVENT)
    enqueue1(signals[ref++]);

}
#endif

/* returning NOEVENT means that the queue is empty. Result is
 * returned in CurrEvent.tag (global variable). That is the way you
 * do it in this funny language to conserve space ...
 */
void dequeue1(void) {
  if (qend != qbeg) {   /* non empty */

    CurrEvent.tag = queue[qbeg];
    if (qbeg == QUEUE_LENGTH - 1)
```

```
      qbeg = 0;
    else
      qbeg++;
  } else
    CurrEvent.tag = NOEVENT;
}

#endif /* SCOPE_SIGNALQUEUE */

/********************
 * Implementation of The Interpreter *
 ********************/


/* entering states. Invariant is that if called on the whole
 * hierarchy leaves vector of target empty. If called on the part
 * removes the states belonging to this part. This is deep entering,
 * so it sometimes has to search through the hierarchy to decide if
 * the state should be entered.
 */
static void enterand(andstref scope);

static void enteror(orstref scope) {
  tgtsiter j = 0;
  const    horcell *i = GET_OR(scope);
  andstref min = *(GET_AND(*i) + 1);
  const    horcell *max = i;
  andstref andst = 0;

  assert(!activeor_next(scope));
  assert(*i != STMRK);
  VERBOSE_OR( "Entering ", scope );

  while (*max < *(max + 1))
    ++max;

  assert(*max != STMRK);
  assert(*max != HMRK);
  assert(is_andstate(*max));
  assert(is_andstate(min));

  for (; j < tgts_size; ++j)

    if (targets[j] <= *max && targets[j] >= min) {
      for (; *i < targets[j]; ++i);
      assert(i <= max);
      andst = *i;
      assert(is_andstate(andst));
      break;
```

```
      }

  if (andst) {
    if (targets[j] == andst)
      targets[j] = targets[--tgts_size];
  } else {
    /* enter by default: history or initial */
#ifdef SCOPE_HISTORY
    if (*(max + 1) == HMRK)
      andst = history[*(max + 2)];
    else
#endif
      andst = *GET_OR(scope);
  }
  exec(GET_ENTER(andst));
#ifndef SCOPE_CONFSET
  assert (andst != STMRK);
  assert (andst != ROOT_STATE);
  assert (is_andstate(andst));
  assert (scope < STATE_WIDTH);
  next_conf[scope] = andst;
#endif
  enterand(andst);

  assert( activeor_next(scope) );
}

static void enterand(andstref scope)
{
  const handcell *i = GET_AND(scope) + 1;

  assert(is_andstate(scope));
  VERBOSE_AND( "Entering ", scope );

  if (*i != scope) {
    ++i;
    do {
      enteror(*i);
      ++i;
    }
    while (*(i - 1) < *i);
  }
#ifdef SCOPE_CONFSET
  else {
    /* add scope to next configuration */
    assert(neco_size < STATE_WIDTH);
    next_conf[neco_size++] = scope;
  }
#endif
```

```
  assert(scope == ROOT_STATE || activeand_next(scope));

}

#ifdef IMPURE_EXIT

static void exitor_impure(orstref);

static void exitand_impure(andstref s)
{
  const handcell *i = GET_AND(s) + 1;

  VERBOSE_AND( "Exiting-and ", s );

  assert(is_andstate(s));
  assert(activeand_next(s));

  if (*i != s) {
    ++i;
    do {
      exitor_impure(*i);
      ++i;

    } while (*(i - 1) < *i);
  }
#ifdef SCOPE_CONFSET
  else {
    /* remove s from configuration */
    confiter c = neco_size - 1;
    assert(conf_member_ext(next_conf, s));

    for (; next_conf[c] != s; --c) {
      assert(c > 0);
    }
    next_conf[c] = next_conf[--neco_size];
  }
  assert(!activeand_next(s));
#endif /* SCOPE_CONFSET */
}

#else /* not IMPURE_EXIT */

#define exitor_impure(s)  exitor_pure(s)

#endif


#ifdef IMPURE_EXIT
```

```
static void exitor_impure(const orstref scope)
{
#ifdef SCOPE_CONFSET
  confiter j;
  const horcell *i = GET_OR(scope);
  andstref min = *(GET_AND(*i) + 1);
  const horcell *max = i;
  confiter oldbufsize = neco_size;
#endif
  andstref andst;

  VERBOSE_OR("Exiting-or-impure ", scope );
  assert( activeor_next(scope) );

#ifdef SCOPE_CONFSET
  while (*max < *(max + 1))
    ++max;
  assert(*i != STMRK);
  assert(*max != STMRK);
  assert(*max != HMRK);
  assert(is_andstate(*max));
  assert(is_andstate(min));

  for (j = 0; j < neco_size; ++j)

    if (next_conf[j] <= *max && next_conf[j] >= min) {
      for (; *i < next_conf[j]; ++i);
      assert(i <= max);
      andst = *i;
      break;
    }

#if defined(SCOPE_HISTORY)
  if (*(max + 1) == HMRK)
    history[*(max + 2)] = andst;
#endif

#else
  assert (scope < STATE_WIDTH);
  andst = next_conf[scope];

#if defined(SCOPE_HISTORY)
  {
    const horcell *i = GET_OR(scope);
    do ++i;
    while (*(i-1) < *i);
    if (*i == HMRK)
      history[*(i + 1)] = andst;
  }
```

```
#endif

#endif


  assert(andst != ROOT_STATE);
  assert(andst != STMRK );
  assert(andst != HMRK );
  assert(is_andstate(andst));
  assert(activeand_next(andst));

  exitand_impure(andst);

#ifndef SCOPE_CONFSET /* remove from next configuration */
  assert(scope < STATE_WIDTH);
  next_conf[scope] = STMRK;
  assert(!activeand_next(andst));
#endif
  exec(GET_EXIT(andst));
#ifdef SCOPE_CONFSET
  assert(oldbufsize >= neco_size);
#endif
  assert(!activeor_next(scope));

  VERBOSE_OR("Exited-or-impure ", scope );

} /* exitor_impure() */

#endif      /* IMPURE_EXIT */

#ifndef PURE_EXIT

#define exitor_pure(s) exitor_impure(s)

#else   /* defined(PURE_EXIT) */

#ifndef SCOPE_CONFSET
#error "Pure exit only works in confset mode"
#endif

void exitor_pure(orstref s)
{
#ifndef NDEBUG
  int removed = 0;
  int presize = neco_size;
#endif
  confiter j = 0;
  const andstref *const i = GET_OR(s);
  andstref min = *(GET_AND(*i) + 1);
```

```
  const horcell *max = i;

  VERBOSE_OR( "Exiting-or-pure ", s );
  assert(is_orstate(s));
  assert(!conf_empty(next_conf));
  assert(activeor_next(s));

  while (*max < *(max + 1))
    ++max;

  assert(is_andstate(*max));
  assert(ROOT_STATE > *max || ROOT_STATE < min);

  while (j < neco_size)
    if (next_conf[j] >= min && next_conf[j] <= *max) {
      next_conf[j] = next_conf[--neco_size];
#ifndef NDEBUG
      removed++;
#endif
    } else
      j++;

  assert(removed);
  assert(neco_size + removed == presize);
  assert(!activeor_next(s));
}

#endif     /* PURE_EXIT */

/* Fire a transition. Transition is assumed to be enabled.
 * The parameter passed is assumed to be the index in
 * transitions array were the guard/action indicators are kept.
 * We point into the middle of the record for slight efficiency
 * gain. The preceding part will not be used in this function
 * anyway. The assumption is that targets are empty on entry
 * and on exit.
 */

void fire(const transcell * ptr)
{
  actionref ac = STMRK;
  const transcell *s;

#ifdef SCOPE_GUARDS
  guardref gd = STMRK;

  UNMANGLE_ACGD(ptr, ac, gd);
  if (!eval(gd))
    return;
```

```
#else
  UNMANGLE_AC(ptr, ac);
#endif
  assert(targets_empty());

  s = ptr;

  while (*s != STMRK) {

    assert(*s < MARKNO);

#ifndef IMPURE_EXIT
    assert(*s != IMRK && *s != HMRK);
#endif
#ifndef PURE_EXIT
    assert(*s != FPSCOP && *s != NFPSCOP);
#endif

    switch (*(s++)) {
#ifdef IMPURE_EXIT
    case HMRK:  /* flat non pure exit */

      do
        exitor_impure(*(GET_AND(*(s++))));
      while (*s >= MARKNO);
      break;

    case IMRK:  /* nonflat and nonpure exit */

      exitor_impure(*(s++));
      while (*s >= MARKNO)
        s++;
      break;
#endif
#ifdef PURE_EXIT
    case FPSCOP:  /* flat pure exit */

      do
        exitor_pure(*(GET_AND(*(s++))));
      while (*s >= MARKNO);
      break;

    case NFPSCOP:  /* nonflat and pure exit */

      exitor_pure(*(s++));
      while (*s >= MARKNO)
        s++;
#endif
    }
```

```
      assert(*s < MARKNO);
  }

  VERBOSE("TRANAC:")
  exec(ac);
  VERBOSE(" TRANAC\n")

  while (*ptr != STMRK) {

      assert(*ptr < MARKNO);

#ifndef IMPURE_EXIT
      assert(*s != IMRK && *s != HMRK);
#endif

#ifndef PURE_EXIT
      assert(*s != FPSCOP && *s != NFPSCOP);
#endif

      if (
#ifdef IMPURE_EXIT
          *ptr == HMRK ||
#endif
#ifdef PURE_EXIT
          *ptr == FPSCOP
#else
          0
#endif
          ) {     /* flat */

        while (*(++ptr) >= MARKNO) {
          targets[0] = *ptr;
          tgts_size = 1;
          enteror(*(GET_AND(*ptr)));
        }

      } else {  /* non flat */

        orstref scope = *(++ptr);

        for (++ptr; *ptr >= MARKNO; ++ptr)
          targets[tgts_size++] = *ptr;
        enteror(scope);
      }

      assert(targets_empty());
      assert(*ptr < MARKNO);
  }
```

```
}

/* microstep - assumes that proper event is already in CurrEvent no
 * conflict resolution currently. Everything will collapse
 * on conflict
 */
void microstep(void)
{

  const transcell *t = trans + tranidx[CurrEvent.tag];

  assert(CurrEvent.tag < NOEVENT);

  if (tranidx[CurrEvent.tag] != TRANS_MAX)
    do {

      unsigned int pc, nc;

      UNMANGLE_PCNC(t, pc, nc);

      for (; pc > 0; --pc, ++t)
        if (!activeand(*t))
          goto skiptr;

      for (; nc > 0; --nc, ++t)
        if (activeand(*t))
          goto skiptr;

      fire(t);  /* checking guard is deferred to fire */

          skiptr:while (*t != STMRK)
        ++t;
      ++t;

    }
    while (t < trans + tranidx[CurrEvent.tag + 1]);
}

/* assumes that external event is already in CurrEvent the newest
 * configuration is always kept in next_conf. prev_conf is only
 * occasionally used for guard computations.
 */
void macrostep(void)
{

#ifdef SCOPE_SIGNALQUEUE
  while (CurrEvent.tag != NOEVENT) {
#endif
    confiter i = 0;
```

```
#ifdef SCOPE_CONFSET
    prco_size = neco_size;
    for (; i < neco_size; ++i)
       prev_conf[i] = next_conf[i];
#else
    for (; i < STATE_WIDTH; ++i)
       prev_conf[i] = next_conf[i];
#endif

    assert(!conf_empty(next_conf));

    microstep();

#ifdef SCOPE_SIGNALQUEUE
    dequeue1();
  }
#endif
}

/* initialize the model. Note: st_init will only work on empty
 * configuration.  If there is a need for multiple initializations
 * at runtime another function (see testdrv) has to be called to
clean
 * current conf
 */
void st_init(void)
{
#ifdef SCOPE_CONFSET
  prco_size = neco_size = 0;
#else
  confiter i = 0;
  for( i; i < STATE_WIDTH; ++i )
    prev_conf[i] = next_conf[i] = STMRK;
#endif
  enterand(ROOT_STATE);

#ifdef SCOPE_SIGNALQUEUE
  dequeue1();
  macrostep();
#endif
}
```

Appendix C

SCOPE Flat Engine

```
/* File : CodGenCF.c
 * Comment: Runtime interpreter for programs generated with CodGenCF
 *
 * Copyright (C) 2003 Andrzej Wasowski
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation; either version 2 of
 * the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA
 * 02111-1307, USA.
 */

/* a hack to compile an avr version easily (unfortunately no assert
 * in avr lib, at least in the one I got installed)
 */

#ifdef NDEBUG
#define assert(p)
#else
#include <assert.h>
#endif

#define SCOPE_VERBOSE 0

#if (SCOPE_VERBOSE && !defined(NDEBUG))
```

```
    #define VERBOSE(msg)    { printf ( "%s", msg ); }
#else
  #define VERBOSE(msg)
#endif

#ifndef NDEBUG

#include <stdlib.h>
#include <stdio.h>

#if STATE_WIDTH > 0
/* check if conf is empty (used in assertions) */
static int conf_empty(conf c)
{
  int i = 0;
  for (; i < STATE_WIDTH; i++)
    if (c[i] != STMRK)
      return 0;
  return 1;
}

/* check if state s is a member of c */
int conf_member_ext(conf c, stref s)
{
  return c[anatomy[s]] == s;
}

void next_conf_reset(void)
{
  confiter i;
  for (i = 0; i < STATE_WIDTH; ++i)
    next_conf[i] = STMRK;
}

int compar(const void *a, const void *b)
{
  if (*(stref *) a < *(stref*) b)
    return -1;
  else if (*(stref *) a == *(stref *) b)
    return 0;
  return 1;
}

void print_st_sym(stref s)
{
  int c = 0;
  while (1) {
    if (st_syms[c].id == s) {
      printf("%s ", st_syms[c].name);
```

```
      break;
    } else ++c;
  }
}

void print_mchn_sym(mchnref s)
{
  int c = 0;
  while (1) {
    if (mchn_syms[c].id == s) {
      printf( "%s", mchn_syms[c].name);
      break;
    } else ++c;
  }
}

/* check if 'what' is a state. Only works in --debug mode, as it
 * uses symbolic dictionaries
 */
unsigned is_state(stref what)
{
  int c = 0;
  while (st_syms[c].id != 0) {
    if (st_syms[c].id == what)
      return 1;
    c++;
  }
  return 0;
}

/* check if 'what' is an orstate. Only works in --debug mode, as it
 * uses symbolic dictionaries
 */
unsigned is_mchn(mchnref what)
{
  int c = 0;
  while (mchn_syms[c].id != 0) {
    if (mchn_syms[c].id == what)
      return 1;
    c++;
  }
  return 0;
}
#endif /* STATE_WIDTH > 0 */

void dump_conf()
{
#if STATE_WIDTH > 0
  int i = 0;
```

```
  conf myconf;

  for (i = 0; i < STATE_WIDTH; ++i)
    myconf[i] = next_conf[i];

  printf("conf: [");
  qsort(myconf, STATE_WIDTH, sizeof(stref), compar);
  for (i = 0; i < STATE_WIDTH; ++i)
    if (myconf[i] != STMRK)
      print_st_sym(myconf[i]);
  printf("]\n");
#else
  printf("[No discrete states in this model]\n");
#endif
}

#if STATE_WIDTH > 0
VS_BOOL activest_next(const stref what)
{
  assert(is_state (what));
  assert(anatomy[what] < STATE_WIDTH);
  return next_conf[anatomy[what]] == what;
}
#endif

#endif /* NDEBUG */

#if STATE_WIDTH > 0
VS_BOOL active(const stref what)
{
  return prev_conf[anatomy[what]] == what;
}
#endif

#ifdef SCOPE_SIGNALQUEUE

/* Signal Queue - implemented as a round buffer */
static queueiter qbeg = 0;
static queueiter qend = 0;

/* silently assumes that queue is long enough! which shall be
 * the case, since verification checks for that.
 */
void enqueue1(eventtag e)
{

  assert(e < EVENTNO);

  queue[qend] = e;
```

```
  if (qend == QUEUE_LENGTH - 1)
    qend = 0;
  else
    qend++;

  assert(qend != qbeg);
}


#ifdef SCOPE_SLIST
void enqueueList(signalref ref)
{
  while (signals[ref] != NOEVENT)
    enqueue1(signals[ref++]);
}
#endif


/* returning NOEVENT means that the queue is empty result is
 * returned in CurrEvent.tag (global variable) That is the way
 * you do it in this funny language to conserve space ...
 */
void dequeue1(void) {

  if (qend != qbeg) {   /* non empty */

    CurrEvent.tag = queue[qbeg];
    if (qbeg == QUEUE_LENGTH - 1)
      qbeg = 0;
    else
      qbeg++;
  } else
    CurrEvent.tag = NOEVENT;
}


#endif /* SCOPE_SIGNALQUEUE */


/********************
 * Implementation of The Interpreter *
 ********************/


/* Fire a transition. Transition is assumed to be enabled.  The
 * parameter passed is assumed to be the index in transitions array
 * were the guard/action indicators are kept. We point into the
 * middle of the record for slight efficiency gain. The preceding
 * part will not be used in this function anyway. The assumption is
 * that targets are empty on entry and on exit.
 */
void fire(const transcell * ptr)
{
```

```
  actionref ac = STMRK;

#ifdef SCOPE_GUARDS
  guardref gd = STMRK;

  UNMANGLE_ACGD(ptr, ac, gd);
  if (!eval(gd))
    return;
#else
  UNMANGLE_AC(ptr, ac);
#endif
  VERBOSE("TRANAC:")
  exec(ac);
  VERBOSE(" TRANAC\n")
#if STATE_WIDTH > 0
  while (*ptr != STMRK) {
    assert(is_state (*ptr));
    next_conf[anatomy[*ptr]]=*(ptr++);

  }
#else
  assert(*ptr == STMRK); /* no state machines */
#endif
}

/* microstep - assumes that proper event is already in CurrEvent no
 * conflict resolution currently. Everything will collapse
 * on conflict
 */

void microstep(void)
{
  const transcell *t = trans + tranidx[CurrEvent.tag];
  assert(CurrEvent.tag < NOEVENT);
  if (tranidx[CurrEvent.tag] != TRANS_MAX)
    do {

      unsigned int pc, nc;

      UNMANGLE_PCNC(t, pc, nc);
#if STATE_WIDTH > 0
      for (; pc > 0; --pc, ++t)
        if (!active(*t))
          goto skiptr;

      for (; nc > 0; --nc, ++t)
        if (active(*t))
          goto skiptr;
#else
```

```
      assert (pc==0 && nc==0); /* they could be left out... */
#endif

      fire(t);  /* checking guard is deferred to fire */
#if STATE_WIDTH > 0
          skiptr:while (*t != STMRK)
        ++t;
#else
      if (*t != STMRK) ++t;
      assert(*t == STMRK);
#endif
      ++t;
    } while (t < trans + tranidx[CurrEvent.tag + 1]);
}


/* assumes that external event is already in CurrEvent the newest
 * configuration is always kept in next_conf. prev_conf is only
 * occasionally used for guard computations.
 */
void macrostep(void)
{

#ifdef SCOPE_SIGNALQUEUE
  while (CurrEvent.tag != NOEVENT) {
#endif
#if STATE_WIDTH > 0
    confiter i = 0;
    for (; i < STATE_WIDTH; ++i)
      prev_conf[i] = next_conf[i];

    assert(!conf_empty(next_conf));
#endif

    microstep();

#ifdef SCOPE_SIGNALQUEUE
    dequeue1();
  }
#endif
}
```

Appendix D

SCOPE Test Drivers

The following three drivers where used in tests and benchmarks. Remember that the main kernel loop is normally implemented by the developer of the embedded application. The loop typically involves communication with sensor drivers. In benchmarking the code generation we have chosen to keep this loop as simple as possible, to decrease its impact on the result of benchmarks. The first driver, uses a minimal loop, that is just used for size benchmarks. The second driver (CodGenC1_smokedrv) sends long random traces of events to the model. The third driver was used to implement test scripts. Similar drivers where written for IAR visualSTATE, to facilitate fair comparison.

```
/* File   : CodGenC1_tinydrv.c
   Comment: Tiny driver used for code generation *size* comparisons
            with visualSTATE

   Copyright (C) 2001-2002 Andrzej Wasowski

   This program is free software; you can redistribute it and/or
   modify it under the terms of the GNU General Public License as
   published by the Free Software Foundation; either version 2 of
   the License, or (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA
   02111-1307, USA.
*/
```

```
int main (void)
{
  volatile int i = 0;
  st_init();
  while (1) {
    CurrEvent.tag = i;
    macrostep();
  }
  return 0;
}


/* File    : CodGenC1_smokedrv.c
   Comment: Small driver used for testing efficiency

   Copyright (C) 2001-2002 Andrzej Wasowski

   This program is free software; you can redistribute it and/or
   modify it under the terms of the GNU General Public License as
   published by the Free Software Foundation; either version 2 of
   the License, or (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   GNU General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA
   02111-1307, USA.

   NOTE:

   This is extremely similar to tiny driver, but also allows
   parameter for number of iterations. Time is not measured
   (use system "time" command or sthg similar).
*/

#include <stdlib.h>
#include <stdio.h>
#include <time.h>


int init (int argc)
{
  if ( argc > 3 ) {
    printf ("init,");
    fflush(stdout);
  }
```

```
    st_init();

}

int main (int argc,const  char * argv[])
{
  int i = 0;
  int j = 0;
  unsigned max1 = atoi (argv[1]);
  unsigned max2 = atoi (argv[2]) +1;

  printf ("Smoke test. Events %d. Limit %d. Number of tries "
                "%d...\n", EVENTNO, max2, max1);

  srand(time(NULL));
  init( argc );

  /* deliberate code repetition. This way the test is only
   * performed once and does not influence the speed
   * characteristic significantly.
   */
  if ( max2 != 0 )
    for(; i<max1; ++i) {

      if (rand() % 100 == 0)
        init( argc );

      j = rand();

      CurrEvent.tag = j % max2;
      if ( argc > 3 ) {
        printf ("%d,", CurrEvent.tag);
        fflush(stdout);
      }
      macrostep();
    }
  else for(; i < max1; ++i) {
      init( argc );
      if ( argc > 3 ) {
        printf ("%d,", CurrEvent.tag);
        fflush(stdout);
      }
      macrostep();
    }
  printf ("done\n");
  return 0;
}
```

```
/* File   : CodGenC1_testdrv.c
   Comment: Tiny driver used for testing of C1/CF

   Copyright (C) 2001-2002 Andrzej Wasowski

   This program is free software; you can redistribute it and/or
   modify it under the terms of the GNU General Public License as
   published by the Free Software Foundation; either version 2 of
   the License, or (at your option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   General Public License for more details.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA
   02111-1307, USA.
*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* defined in CodGenC1.c in DEBUG mode */
extern void dump_conf(void);
extern const char * evdict[];

/* This function is basically an interpreter for a small test
 * language, with syntax is as follows
 *
 * cmd -> init | conf | macro e | quit
 *
 * where
 *
 * init, conf, macro and quit are kewywords and e is an event name.
 * Semantics:
 *
 * init - initialize the model
 * conf - print current configuration on stdout (no changes to the
 *        model)
 * quit - exit the program
 * macro e - make e a current event and start a macrostep.
 * autoconf - enters autoconf mode. In this mode configuration is
 *            printed automatically after processing of each
 *            command.
 */
int main(void) {
```

```c
  int autoconf = 0;
  /* unsafe but this is only a test program */
  char buf[400];
  int i;

  while (1) {

    if (scanf( "%s", buf )==EOF)
      exit(EXIT_FAILURE);

    else if ( strcmp( buf, "autoconf" ) == 0 )
      autoconf = 1;

    else if ( strcmp( buf, "quit" ) == 0 )
      exit(EXIT_SUCCESS);

    else if ( strcmp( buf, "conf" ) == 0 ) {
      if (!autoconf)
        dump_conf();

    } else if ( strcmp( buf, "init" ) == 0 ) {
      st_init();

    } else if (strcmp ( buf, "macro" ) == 0 ) {

      if (scanf ( "%s", buf ) == EOF)
        exit (EXIT_SUCCESS);

      for (i = 0; i < EVENTNO; i++) {
        if (strcmp(evdict[i],buf) == 0) {
          CurrEvent.tag = i;
          macrostep();
          break;
        }
      }
      if ( i == EVENTNO ) {

        fprintf(stderr, "Unknown event.\n");
        exit (EXIT_FAILURE);
      }
    } else {
      fprintf (stderr, "Syntax error.\n");
      exit(EXIT_FAILURE);
    }
    if (autoconf) dump_conf();
  }
  return 0;
}
```

Appendix E

An Example of SCOPE Generated Code

E.1   A Simple Controller Model
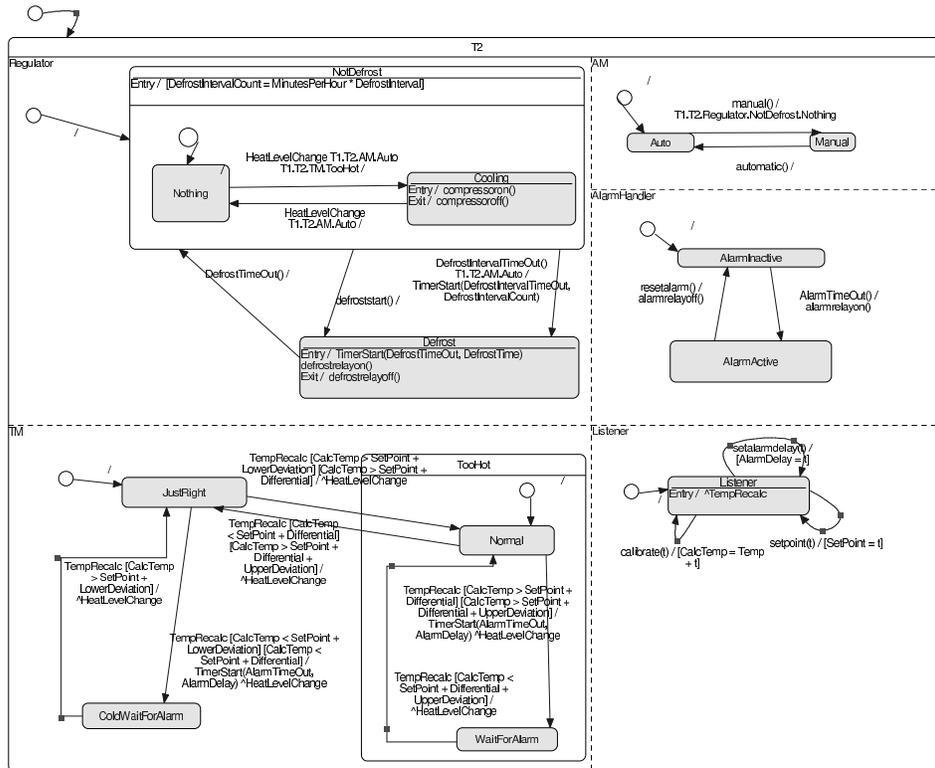
E.2   Hierarchy Tree

Generated using SCOPE and visualized using graphviz:
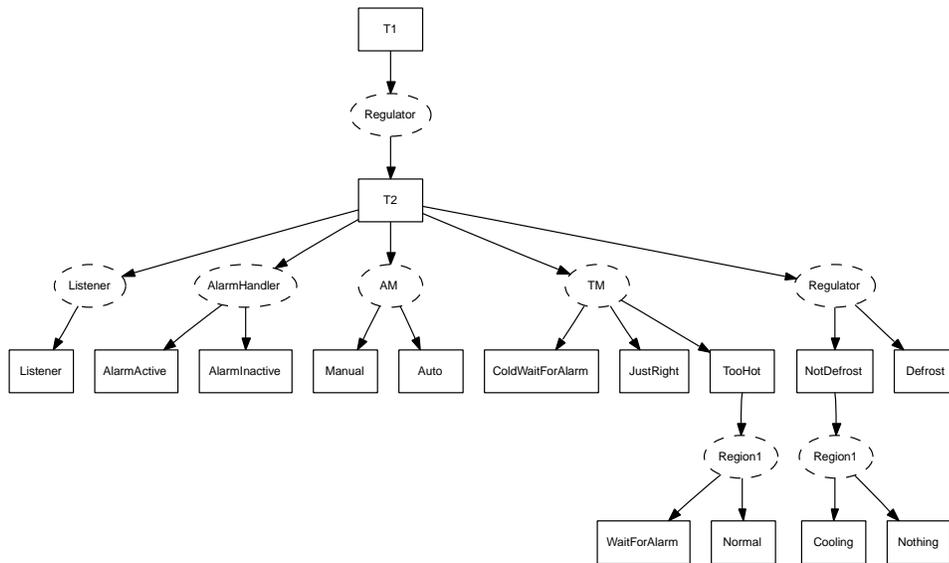
E.3 Hierarchical Encoding

```
/***
 * This is a generated file! Don't edit it manually.
 * File name      : ekc-c1.h
 * System name    : EKC
 * Source file    : Project.vsp
 * Generation time: 1126125273.525
 * Code generator : CodGenC1
 **/

/* fixed width integer types */

#include <VSTypes.h>

/* API interface */

void st_init(void);
void macrostep(void);

/* inferred type for referring signals and events */

typedef VS_UINT8 eventtag;

/* event names */

#define AlarmTimeOut  0
#define DefrostIntervalTimeOut  1
#define DefrostTimeOut  2
#define TimeTick  3
#define automatic  4
#define calibrate  5
#define defroststart  6
#define manual  7
#define resetalarm  8
#define setalarmdelay  9
#define setpoint  10

/* event types */
struct Structcalibrate {
  VS_INT8 f0;
};

struct Structsetalarmdelay {
  VS_UINT8 f0;
};

struct Structsetpoint {
  VS_INT8 f0;
```

```
};

/* union of event types */
struct StructEvent {
  eventtag tag;
  union {
    struct Structsetpoint _E_setpoint;
    struct Structsetalarmdelay _E_setalarmdelay;
    struct Structcalibrate _E_calibrate;
  } fields;

};

extern struct StructEvent CurrEvent;

/* model constants */
#define DefrostInterval  1
#define DefrostTime  15
#define Differential  2
#define LowerDeviation  -10
#define MinutesPerHour  10
#define Temp  20
#define UpperDeviation  10

/* variable declarations (none) */

/* number of events */
#define EVENTNO  13
#define NOEVENT  EVENTNO
```

```
/***
 * This is a generated file! Don't edit it manually.
 * File name      : ekc-c1.c
 * System name    : EKC
 * Source file    : Project.vsp
 * Generation time: 1126125273.547
 * Code generator : CodGenC1
 **/

/* include public information */
#include "ekc-c1.h"

/* prototypes for functions used in actions/guards */
#include "ekc_extern.h"

/* Mnemonics for LR-bytecode (used as comments) in arrays */
#define HIST
#define EVNT        /* event reference */
#define ENEX(n)       /* entry/exit ref  */
#define ACGD(n)       /* action/guard    */
#define PCNC(n)       /* pos/neg-cond    */
#define ORST      /* or-state ref    */
#define ANDST       /* and-state ref   */

/* inferred type of action refs */
typedef VS_UINT8 actionref;

/* inferred type of guard refs */
typedef VS_UINT8 guardref;

/* inferred type of or-state ref */
typedef VS_UINT8 orstref;

/* inferred type of and-state ref */
typedef VS_UINT8 andstref;

/* inferred return type for all actions */
typedef VS_UINT8 signalref;

/* inferred type for refering to trans */
typedef VS_UINT8 tranref;

/* type of iterators over trans (|traniter| >= |tranref|) */
typedef VS_UINT8 traniter;

/* type of configurations */
typedef andstref conf[10];

/* type for integer iterators over configuration arrays */
```

```
typedef VS_UINT8 confiter;

/* type for integer iterators over sets of targets */
typedef VS_UINT8 tgtsiter;

/* type of integer iterators over signal queue elements */
typedef VS_UINT8 queueiter;

/* model-independent types (here for simplicity) */
typedef andstref horcell;
typedef orstref handcell;
typedef andstref transcell;
typedef andstref historyref;

/* type of symbols suppressed in release mode */

/* preprocessor constants for bytecode markers */
#define STMRK  0
#define HMRK   1
#define IMRK   2
#define FPSCOP  3
#define NFPSCOP  4
#define MARKNO  5

/* signal queue controlling macros and declarations */
#define SCOPE_SIGNALQUEUE
#define QUEUE_LENGTH  10
static eventtag queue[QUEUE_LENGTH] =  {
  /*0*/ NOEVENT, NOEVENT, NOEVENT, NOEVENT, NOEVENT, NOEVENT,
  /*6*/ NOEVENT, NOEVENT, NOEVENT, NOEVENT,
};
#define SET_RETURN_SIGNAL(s)  {_vssignal = (s);}

/* variable definitions */
static VS_UINT8 AlarmDelay = 30;
static VS_INT8 CalcTemp = 0;
static VS_UCHAR DefrostCount = 0;
static VS_UINT8 DefrostIntervalCount = 0;
static VS_INT8 SetPoint = 3;

/* current event holder */
struct StructEvent CurrEvent = { EVENTNO };

/* history related control (switch and vector if needed) */
#undef SCOPE_HISTORY

/* prievious configuration */
static conf prev_conf =  {
  /*0*/ STMRK, STMRK, STMRK, STMRK, STMRK, STMRK, STMRK,
```

```
  /*7*/ STMRK, STMRK, STMRK,
};

/* next configuration */
static conf next_conf =  {
  /*0*/ STMRK, STMRK, STMRK, STMRK, STMRK, STMRK, STMRK,
  /*7*/ STMRK, STMRK, STMRK,
};

/* targets pool */
static conf targets =  {
  /* 0 */ STMRK,
};

/* or-state projection of hierarchy */
const horcell hor[18] =  {
  /*  0 */ HMRK, HMRK, ANDST 45, ANDST 43, ANDST 37,
  /*  5 */ ANDST 39, ANDST 33, ANDST 35, ANDST 22, ANDST 28,
  /* 10 */ ANDST 31, ANDST 24, ANDST 26, ANDST 14, ANDST 20,
  /* 15 */ ANDST 5, ANDST 10, STMRK,
};

/* dictionary for accessing hor */
const VS_UINT8 hor_dict[10] =  {
  /*  0 */ 0, 0, 2, 3, 4, 6, 8, 11, 13, 15,
};

/* macro for accessing hor */
#define GET_OR(s)  (hor+ hor_dict[(s)] )

/* and-state projection of hierarchy */
const handcell hand[56] =  {
  /*  0 */ HMRK, HMRK, HMRK, HMRK, HMRK,
  /*  5 */ ORST 9, ANDST 5, STMRK, ENEX(2) 11, 10,
  /* 10 */ ORST 9, ANDST 10, STMRK, ENEX(1) 9, ORST 8,
  /* 15 */ ANDST 5, ORST 9, STMRK, ENEX(2) 8, 7,
  /* 20 */ ORST 8, ANDST 20, ORST 6, ANDST 22, ORST 7,
  /* 25 */ ANDST 24, ORST 7, ANDST 26, ORST 6, ANDST 24,
  /* 30 */ ORST 7, ORST 6, ANDST 31, ORST 5, ANDST 33,
  /* 35 */ ORST 5, ANDST 35, ORST 4, ANDST 37, ORST 4,
  /* 40 */ ANDST 39, STMRK, ENEX(1) 6, ORST 3, ANDST 43,
  /* 45 */ ORST 2, ANDST 5, ORST 3, ORST 4, ORST 5,
  /* 50 */ ORST 6, ORST 8, STMRK, ANDST 5, ORST 2,
  /* 55 */ STMRK,
};

/* hand is accessed directly. */

/* macro for accessing hand */
```

```
#define GET_AND(s)  (hand+ (s))

/* determine if model uses signal lists */
#undef SCOPE_SLIST

/* define if configuration set implemented using lists
   (flags otherwise) */
#undef SCOPE_CONFSET

/* activate needed exit modes */
#undef PURE_EXIT
#define IMPURE_EXIT

/* no multiple signals on single transition */

/* action dispatcher */
void exec ( const actionref a ) {
  #ifdef SCOPE_SIGNALQUEUE
  signalref _vssignal = NOEVENT;
  extern void enqueue1 ( eventtag );
#ifdef SCOPE_SLIST
  extern void enqueueList ( signalref );
#endif
  #endif

  switch (a) {
  case 0:
  case 1:
  case 2:
  case 3:
  case 4:
  case 5:
    break;
  case 6:
    SET_RETURN_SIGNAL(11);
    break;
  case 7:
    TimerStart(DefrostTime,DefrostTimeOut);
    defrostrelayon();
    break;
  case 8:
    defrostrelayoff();
    break;
  case 9:
    DefrostIntervalCount = (MinutesPerHour)*(DefrostInterval);
    break;
  case 10:
    compressoron();
    break;
```

```
  case 11:
    compressoroff();
    break;
  case 12:
    AlarmDelay = CurrEvent.fields._E_setalarmdelay.f0;
    break;
  case 13:
    CalcTemp = (Temp)+(CurrEvent.fields._E_calibrate.f0);
    break;
  case 14:
    SetPoint = CurrEvent.fields._E_setpoint.f0;
    break;
  case 15:
    alarmrelayoff();
    break;
  case 16:
    alarmrelayon();
    break;
  case 17:
    SET_RETURN_SIGNAL(12);
    break;
  case 18:
    TimerStart(AlarmDelay,AlarmTimeOut);
    SET_RETURN_SIGNAL(12);
    break;
  case 19:
    TimerStart(DefrostIntervalCount,DefrostIntervalTimeOut);
    break;
  }

  #ifdef SCOPE_SIGNALQUEUE
  #ifdef SCOPE_SLIST
  if (_vssignal < NOEVENT) enqueue1(_vssignal);
  else if (_vssignal > NOEVENT)
    enqueueList (_vssignal-NOEVENT-1);
  #else
  if (_vssignal != NOEVENT) enqueue1(_vssignal);
  #endif
  #endif
}

/* guards dispatcher */
int eval ( const guardref g ) {
  switch (g) {
  case 5:
    return (((CalcTemp)>((SetPoint)+(Differential)))&&((CalcTemp)>
           (((SetPoint)+(Differential))+(UpperDeviation))));
  case 6:
    return
```

```
((CalcTemp)<(((SetPoint)+(Differential))+(UpperDeviation))));
  case 7:
    return (((CalcTemp)<((SetPoint)+(LowerDeviation)))&&((CalcTemp)<
           ((SetPoint)+(Differential))));
  case 8:
    return (((CalcTemp)>((SetPoint)+(LowerDeviation)))&&((CalcTemp)>
           ((SetPoint)+(Differential))));
  case 9:
    return (((CalcTemp)<((SetPoint)+(Differential)))&&((CalcTemp)>
           (((SetPoint)+(Differential))+(UpperDeviation))));
  case 10:
    return ((CalcTemp)>((SetPoint)+(LowerDeviation)));
  case 11:
    return (1);
  }
  return 1;
}

/* transitions array */
#define TRANS_MAX   143

const transcell trans[TRANS_MAX] =  {
  /*   0 */ PCNC(2) 1, 0, ANDST 37, ACGD(1) 16, HMRK,
  /*   5 */ ANDST 39, STMRK, PCNC(2) 2, 0, ANDST 33,
  /*  10 */ ANDST 14, ACGD(1) 19, HMRK, ANDST 20, STMRK,
  /*  15 */ PCNC(2) 1, 0, ANDST 20, HMRK, ANDST 14,
  /*  20 */ STMRK, PCNC(2) 1, 0, ANDST 35, HMRK,
  /*  25 */ ANDST 33, STMRK, PCNC(2) 1, 0, ANDST 43,
  /*  30 */ ACGD(1) 13, HMRK, ANDST 43, STMRK, PCNC(2) 1,
  /*  35 */ 0, ANDST 14, HMRK, ANDST 20, STMRK,
  /*  40 */ PCNC(2) 2, 0, ANDST 33, ANDST 14, HMRK,
  /*  45 */ ANDST 35, ANDST 5, STMRK, PCNC(2) 1, 1,
  /*  50 */ ANDST 33, ANDST 14, HMRK, ANDST 35, IMRK,
  /*  55 */ ORST 8, ANDST 5, STMRK, PCNC(2) 1, 0,
  /*  60 */ ANDST 39, ACGD(1) 15, HMRK, ANDST 37, STMRK,
  /*  65 */ PCNC(2) 1, 0, ANDST 43, ACGD(1) 12, HMRK,
  /*  70 */ ANDST 43, STMRK, PCNC(2) 1, 0, ANDST 43,
  /*  75 */ ACGD(1) 14, HMRK, ANDST 43, STMRK, PCNC(2) 1,
  /*  80 */ 0, ANDST 24, ACGD(2) 18, 11, HMRK,
  /*  85 */ ANDST 26, STMRK, PCNC(2) 1, 0, ANDST 26,
  /*  90 */ ACGD(2) 17, 10, HMRK, ANDST 24, STMRK,
  /*  95 */ PCNC(2) 1, 0, ANDST 22, ACGD(2) 18, 9,
  /* 100 */ HMRK, ANDST 31, STMRK, PCNC(2) 1, 0,
  /* 105 */ ANDST 22, ACGD(2) 17, 8, IMRK, ORST 6,
  /* 110 */ ANDST 24, STMRK, PCNC(2) 1, 0, ANDST 24,
  /* 115 */ ACGD(2) 17, 7, HMRK, ANDST 22, STMRK,
  /* 120 */ PCNC(2) 1, 0, ANDST 31, ACGD(2) 17, 6,
  /* 125 */ HMRK, ANDST 22, STMRK, PCNC(2) 3, 0,
  /* 130 */ ANDST 28, ANDST 33, ANDST 5, HMRK, ANDST 10,
```

```
  /* 135 */ STMRK, PCNC(2) 2, 0, ANDST 33, ANDST 10,
  /* 140 */ HMRK, ANDST 5, STMRK,
};

/* dictionary of transition lists */
const tranref tranidx[14] =  {
  /*  0 */ 0, 7, 15, 143, 21, 27, 34, 40, 58, 65,
  /* 10 */ 72, 79, 128, 143,
};

/* symbolic names of or-states supressed in release mode */

/* symbolic names of and-states supressed in release mode */

/* symbolic names of events left out in release mode. */

/* reference to topmost and-state */
#define ROOT_STATE  52

/* bound on configuration size */
#define STATE_WIDTH  10

/* bound on target width */
#define TGT_WIDTH  1

/* macro for accessing encoded pos/neg counters */
#define UNMANGLE_PCNC(ptr,i,j)  { { (i) = + (*((ptr)++)); } \
        { (j) = + (*((ptr)++)); } }

/* macro for accessing encoded ac/gd references */
#define UNMANGLE_ACGD(ptr,i,j)  { (i) = *ptr; if (i >= MARKNO) \
          (ptr)++; (j) = *ptr; \
          if ( (j) >= MARKNO) (ptr)++;}

/* get enter action for state s */
#define GET_ENTER(s)  ((*(GET_AND(s)-2) == STMRK \
      ||(*(GET_AND(s)-3) == STMRK )) \
      ?*(GET_AND(s)-1) : STMRK)

/* get exit action for state s */
#define GET_EXIT(s)  ((*(GET_AND(s)-3) == STMRK ) \
      ?*(GET_AND(s)-2) : STMRK)

/* determine if arbitrary guards are used */
#define SCOPE_GUARDS

/* runtime interpreter */
#include <CodGenC1.c>
```

E.4   Flat Encoding

```
/***
 * This is a generated file! Don't edit it manually.
 * File name      : ekc-cf.h
 * System name    : EKC
 * Source file    : Project.vsp
 * Generation time: 1126125278.289
 * Code generator : CodGenCF
 **/

/* fixed width integer types */
#include <VSTypes.h>

/* API interface */
void st_init(void);
void macrostep(void);

/* inferred type for referring signals and events */
typedef VS_UINT8 eventtag;

/* event names */
#define AlarmTimeOut  0
#define DefrostIntervalTimeOut  1
#define DefrostTimeOut  2
#define TimeTick  3
#define automatic  4
#define calibrate  5
#define defroststart  6
#define manual  7
#define resetalarm  8
#define setalarmdelay  9
#define setpoint  10

/* event types */
struct Structcalibrate {
  VS_INT8 f0;
};

struct Structsetalarmdelay {
  VS_UINT8 f0;
};

struct Structsetpoint {
  VS_INT8 f0;
};

/* union of event types */
struct StructEvent {
```

```
    eventtag tag;
    union {
        struct Structsetpoint _E_setpoint;
        struct Structsetalarmdelay _E_setalarmdelay;
        struct Structcalibrate _E_calibrate;
    } fields;
};

extern struct StructEvent CurrEvent;

/* model constants */
#define DefrostInterval   1
#define DefrostTime   15
#define Differential   2
#define LowerDeviation   -10
#define MinutesPerHour   10
#define Temp   20
#define UpperDeviation   10

/* variable declarations */

/* number of events */
#define EVENTNO   13
#define NOEVENT   EVENTNO
```

```
/***
 * This is a generated file! Don't edit it manually.
 * File name      : ekc-cf.c
 * System name    : EKC
 * Source file    : Project.vsp
 * Generation time: 1126125278.299
 * Code generator : CodGenCF
 **/

/* include public information */
#include "ekc-cf.h"

/* prototypes for functions used in actions/guards */
#include "ekc_extern.h"

/* Mnemonics for LR-bytecode (used as comments) in arrays */
#define EVNT        /* event reference */
#define ACGD(n)      /* action/guard    */
#define AC(n)       /* action(no guard)*/
#define PCNC(n)      /* pos/neg-cond    */
#define MCHN       /* machine ref      */
#define STATE       /* basic state ref */

/* inferred type of action refs */
typedef VS_UINT8 actionref;

/* inferred type of guard refs */
typedef VS_UINT8 guardref;

/* inferred type of or-state ref */
typedef VS_UINT8 mchnref;

/* inferred type of and-state ref */
typedef VS_UINT8 stref;

/* inferred return type for all actions */
typedef VS_UINT8 signalref;

/* inferred type for refering to trans */
typedef VS_UINT8 tranref;

/* type of iterators over trans (|traniter| >= |tranref|) */
typedef VS_UINT8 traniter;

/* type of configurations */
typedef stref conf[6];

/* type for integer iterators over configuration arrays */
typedef VS_UINT8 confiter;
```

```
/* type for integer iterators over sets of targets */

    /** unused in CodGenCF **/

/* type of integer iterators over signal queue elements */
typedef VS_UINT8 queueiter;

/* model-independent types (here for simplicity) */
typedef mchnref anatomycell;
typedef stref transcell;

/* type of symbols suppressed in release mode */

/* preprocessor constants for bytecode markers */
#define STMRK  0

/* signal queue controlling macros and declarations */
#define SCOPE_SIGNALQUEUE
#define QUEUE_LENGTH  10
static eventtag queue[QUEUE_LENGTH] =  {
  /*  0 */ NOEVENT, NOEVENT, NOEVENT, NOEVENT, NOEVENT, NOEVENT,
  /*  6 */ NOEVENT, NOEVENT, NOEVENT, NOEVENT,
};
#define SET_RETURN_SIGNAL(s)  {_vssignal = (s);}

/* variable definitions */
static VS_UINT8 AlarmDelay = 30;
static VS_INT8 CalcTemp = 0;
static VS_UCHAR DefrostCount = 0;
static VS_UINT8 DefrostIntervalCount = 0;
static VS_INT8 SetPoint = 3;

/* current event holder */
struct StructEvent CurrEvent = { EVENTNO };

/* history related control (switch and vector if needed) */

    /** unused in CodGenCF **/

/* prievious configuration */
static conf prev_conf =  {
  /* 0 */ STMRK, STMRK, STMRK, STMRK, STMRK, STMRK,
};

/* next configuration */
static conf next_conf =  {
  /* 0 */ STMRK, STMRK, STMRK, STMRK, STMRK, STMRK,
};
```

```
/* targets pool */

    /** unused in CodGenCF **/

/* or-state projection of hierarchy */

/* hor is accessed directly. */

/* macro for accessing hor */

    /** unused in CodGenCF **/

/* and-state projection of hierarchy */

const anatomycell anatomy[14] =  {
  /*  0 */ STMRK, MCHN 0, MCHN 0, MCHN 4, MCHN 2,
  /*  5 */ MCHN 2, MCHN 3, MCHN 1, MCHN 4, MCHN 5,
  /* 10 */ MCHN 5, MCHN 1, MCHN 2, MCHN 3,
};

/* hand is accessed directly. */

/* macro for accessing hand */

    /** unused in CodGenCF **/

/* determine if model uses signal lists */
#undef SCOPE_SLIST

/* define if configuration set implemented using lists (flags
otherwise) */

    /** unused in CodGenCF **/

/* activate needed exit modes */

    /** unused in CodGenCF **/

/* no multiple signals on single transition */

/* action dispatcher */
void exec ( const actionref a ) {
  #ifdef SCOPE_SIGNALQUEUE
  signalref _vssignal = NOEVENT;
  extern void enqueue1 ( eventtag );
#ifdef SCOPE_SLIST
  extern void enqueueList ( signalref );
#endif
```

```
#endif

switch (a) {
case 0:
case 1:
  AlarmDelay = CurrEvent.fields._E_setalarmdelay.f0;
  SET_RETURN_SIGNAL(11);
  break;
case 2:
  CalcTemp = (Temp)+(CurrEvent.fields._E_calibrate.f0);
  SET_RETURN_SIGNAL(11);
  break;
case 3:
  SetPoint = CurrEvent.fields._E_setpoint.f0;
  SET_RETURN_SIGNAL(11);
  break;
case 4:
  alarmrelayoff();
  break;
case 5:
  alarmrelayon();
  break;
case 6:
  break;
case 7:
  defrostrelayoff();
  DefrostIntervalCount = (MinutesPerHour)*(DefrostInterval);
  break;
case 8:
  compressoroff();
  break;
case 9:
  SET_RETURN_SIGNAL(12);
  break;
case 10:
  TimerStart(AlarmDelay,AlarmTimeOut);
  SET_RETURN_SIGNAL(12);
  break;
case 11:
  TimerStart(DefrostIntervalCount,DefrostIntervalTimeOut);
  TimerStart(DefrostTime,DefrostTimeOut);
  defrostrelayon();
  break;
case 12:
  TimerStart(DefrostTime,DefrostTimeOut);
  defrostrelayon();
  break;
case 13:
  compressoron();
```

```
    break;
  }

  #ifdef SCOPE_SIGNALQUEUE
  #ifdef SCOPE_SLIST
  if (_vssignal < NOEVENT) enqueue1(_vssignal);
  else if (_vssignal > NOEVENT)
    enqueueList (_vssignal-NOEVENT-1);
  #else
  if (_vssignal != NOEVENT) enqueue1(_vssignal);
  #endif
  #endif
}

/* guards dispatcher */
int eval ( const guardref g ) {
  switch (g) {
  case 1:
    return (((CalcTemp)>((SetPoint)+(Differential)))&&((CalcTemp)>
           (((SetPoint)+(Differential))+(UpperDeviation))));
  case 2:
    return
((CalcTemp)<(((SetPoint)+(Differential))+(UpperDeviation)));
  case 3:
    return (((CalcTemp)<((SetPoint)+(LowerDeviation)))&&((CalcTemp)<
           ((SetPoint)+(Differential))));
  case 4:
    return (((CalcTemp)>((SetPoint)+(LowerDeviation)))&&((CalcTemp)>
           ((SetPoint)+(Differential))));
  case 5:
    return (((CalcTemp)<((SetPoint)+(Differential)))&&((CalcTemp)>
           (((SetPoint)+(Differential))+(UpperDeviation))));
  case 6:
    return ((CalcTemp)>((SetPoint)+(LowerDeviation)));
  case 7:
    return (1);
  }
  return 1;
}

/* transitions array */
#define TRANS_MAX  166

const transcell trans[TRANS_MAX] = {
  /*   0 */ PCNC(2) 1, 0, STATE 10, ACGD(2) 5, 1,
  /*   5 */ STATE 9, STMRK, PCNC(2) 3, 0, STATE 11,
  /*  10 */ STATE 3, STATE 1, ACGD(2) 8, 1, STMRK,
  /*  15 */ PCNC(2) 2, 0, STATE 3, STATE 1, ACGD(2) 11,
  /*  20 */ 1, STATE 2, STMRK, PCNC(2) 1, 0,
```

```
  /*  25 */ STATE 2, ACGD(2) 7, 1, STATE 1, STATE 7,
  /*  30 */ STMRK, PCNC(2) 1, 0, STATE 8, ACGD(2) 6,
  /*  35 */ 1, STATE 3, STMRK, PCNC(2) 0, 0,
  /*  40 */ ACGD(2) 2, 1, STMRK, PCNC(2) 2, 0,
  /*  45 */ STATE 11, STATE 1, ACGD(2) 8, 1, STMRK,
  /*  50 */ PCNC(2) 1, 0, STATE 1, ACGD(2) 12, 1,
  /*  55 */ STATE 2, STMRK, PCNC(2) 3, 0, STATE 11,
  /*  60 */ STATE 3, STATE 1, ACGD(2) 8, 1, STMRK,
  /*  65 */ PCNC(2) 2, 0, STATE 3, STATE 1, ACGD(2) 6,
  /*  70 */ 1, STATE 8, STATE 7, STMRK, PCNC(2) 2,
  /*  75 */ 0, STATE 3, STATE 2, ACGD(2) 7, 1,
  /*  80 */ STATE 8, STATE 1, STATE 7, STMRK, PCNC(2) 1,
  /*  85 */ 0, STATE 9, ACGD(2) 4, 1, STATE 10,
  /*  90 */ STMRK, PCNC(2) 0, 0, ACGD(2) 1, 1,
  /*  95 */ STMRK, PCNC(2) 0, 0, ACGD(2) 3, 1,
  /* 100 */ STMRK, PCNC(2) 2, 0, STATE 6, STATE 12,
  /* 105 */ ACGD(2) 10, 7, STATE 13, STMRK, PCNC(2) 2,
  /* 110 */ 0, STATE 13, STATE 12, ACGD(2) 9, 6,
  /* 115 */ STATE 6, STMRK, PCNC(2) 1, 0, STATE 5,
  /* 120 */ ACGD(2) 10, 5, STATE 4, STMRK, PCNC(2) 1,
  /* 125 */ 0, STATE 5, ACGD(2) 9, 4, STATE 12,
  /* 130 */ STATE 6, STMRK, PCNC(2) 2, 0, STATE 6,
  /* 135 */ STATE 12, ACGD(2) 9, 3, STATE 5, STMRK,
  /* 140 */ PCNC(2) 1, 0, STATE 4, ACGD(2) 9, 2,
  /* 145 */ STATE 5, STMRK, PCNC(2) 4, 0, STATE 7,
  /* 150 */ STATE 3, STATE 12, STATE 1, ACGD(2) 13, 1,
  /* 155 */ STATE 11, STMRK, PCNC(2) 3, 0, STATE 11,
  /* 160 */ STATE 3, STATE 1, ACGD(2) 8, 1, STATE 7,
  /* 165 */ STMRK,
};

/* dictionary of transition lists */
const tranref tranidx[14] =  {
  /*  0 */ 0, 7, 23, 166, 31, 38, 43, 57, 84, 91,
  /* 10 */ 96, 101, 147, 166,
};

/* symbolic names of or-states supressed in release mode */

/* symbolic names of and-states supressed in release mode */

/* symbolic names of events left out in release mode. */

/* reference to topmost and-state */

    /** unused in CodGenCF **/

/* bound on configuration size */
#define STATE_WIDTH  6
```

```
/* bound on target width */

    /** unused in CodGenCF **/

/* macro for accessing encoded pos/neg counters */
#define UNMANGLE_PCNC(ptr,i,j)  { { (i) = + (*((ptr)++)); }\
         { (j) = + (*((ptr)++)); } }

/* macro for accessing encoded ac/gd references */
#define UNMANGLE_ACGD(ptr,i,j)   { (i) = *(ptr++); (j) = *(ptr++);}

/* get enter action for state s */

    /** unused in CodGenCF **/

/* get exit action for state s */

    /** unused in CodGenCF **/

/* determine if arbitrary guards are used */
#define SCOPE_GUARDS

/* runtime interpreter */
#include <CodGenCF.c>

/* initial transition */
void st_init(void) {

#if STATE_WIDTH > 0
confiter i = 0;

static const stref iniconf[6] =  {
  /* 0 */ STATE 1, STATE 7, STATE 5, STATE 6, STATE 3,
  /* 5 */ STATE 10,
};
  for (i=0; i < STATE_WIDTH; i++)
    next_conf[i]=iniconf[i];

#endif
  DefrostIntervalCount = (MinutesPerHour)*(DefrostInterval);
  enqueue1( 11 );

#ifdef SCOPE_SIGNALQUEUE
   dequeue1();
  macrostep();
#endif
}
```

E.5   Stub Drivers

The following drivers are empty stubs used in compilation of size-benchmarks and speed benchmarks. SCOPE can also generate stubs that output names of actions to standard output for testing purposes (not shown here).

```
/* This is a generated file!
   Do not edit manually.
   File name: ekc_extern.h */

#include <VSTypes.h>

extern VS_VOID alarmrelayoff(void);
extern VS_VOID alarmrelayon(void);
extern VS_VOID compressoroff(void);
extern VS_VOID compressoron(void);
extern VS_VOID defrostrelayoff(void);
extern VS_VOID defrostrelayon(void);
extern VS_VOID TimerStart(VS_UINT par0, VS_UINT par1);

/* This is a generated file!
   Do not edit it manually.
   File name: ekc_extern.c */

#include <VSTypes.h>
#include "ekc_extern.h"

VS_VOID alarmrelayoff(void) { return; }

VS_VOID alarmrelayon(void) { return; }

VS_VOID compressoroff(void) { return; }

VS_VOID compressoron(void) { return; }

VS_VOID defrostrelayoff(void) { return; }

VS_VOID defrostrelayon(void) { return; }

VS_VOID TimerStart(VS_UINT par0, VS_UINT par1) { return; }
```

Bibliography

[1] Jauhar Ali and Jiro Tanaka. Converting statecharts into Java code. In *5th International Conference on Integrated Design and Process Technology (IDPT'99)*, Dallas,Texas, June 1999. See p. 42

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994. See p. 54

[3] Rajeev Alur, Sampath Kannan, and Mihalis Yannakakis. Communicating hierarchical state machines. In Jirí Wiedermann, Peter van Emde Boas, and Mogens Nielsen, editors, *26th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1644 of *Lecture Notes in Computer Science*, pages 169–178, Prague, Czech Republic, July 1999. Springer-Verlag. See p. 73, 96

[4] Tobias Amnell, Elena Fersman, and Paul Pettersson. Code synthesis for timed automata. *Nordic Journal of Computing*, 9(4), 2003. See p. 54

[5] G. Behrmann, K. G. Larsen, H. R. Andersen, H. Hulgaard, and J. Lind Nielsen. Verification of hierarchical state/event systems using reusability and compositionality. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1579 of *Lecture Notes in Computer Science*, pages 163–177, Amsterdam, The Netherlands, March 1999. Springer-Verlag. See p. 22, 73, 96

[6] Gerd Behrmann, Johan Bengtsson, Alexandre David, Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL implementation secrets. In *Proc. of 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, 2002. See p. 45

[7] Gerd Behrmann, Kaare Kristoffersen, and Kim G.Larsen. Code generation for hierarchical systems. In *NWPT'99 – The 11th Nordic Workshop on Programming Theory*, Uppsala, Sweden, September 1999. See p. 95

[8] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.   See p. 6

[9] Beatrice Bérard, Michel Bidoit, Alain Finkel, Francois Laroussinie, Antoine Petit, Laure Petrucci, Philippe Schnoebelen, and Pierre McKenzie. *Systems and Software Verification. Model-Checking Techniques and Tools.* Springer-Verlag, Berlin-Heidelberg, 2001.   See p. 112

[10] Gérard Berry. The Esterel v5 language primer. version v5_91, July 2000.   See p. 17, 31

[11] Gérard Berry. The foundations of Esterel. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction. Essays in Honour of Robin Milner*, Foundations of Computing Series, pages 425–454. The MIT Press, Cambridge, Massachusetts, 2000.   See p. 124

[12] Robert V. Binder. *Testing Object-Oriented Systems. Models, Patterns and Tools.* Addison-Wesley, 2000.   See p. 73

[13] Dag Björklund, Johan Lilius, and Ivan Porres. Towards efficient code synthesis from statecharts. In Andy Evans, Robert France, and Ana Moreira Bernhard Rumpe, editors, *Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists. Workshop of the pUML-Group.*, Lecture Notes in Informatics P-7, Toronto, Canada, October 2001. GI.   See p. 95

[14] Kirill Bogdanov and Mike Holcombe. Statechart testing method for aircraft control systems. *Software Testing, Verification and Reliability*, 1(11):39–54, 2001.   See p. 54

[15] Kirill Bogdanov and Mike Holcombe. Properties of concurrently taken transitions of Harel statecharts. In *Workshop on Semantic Foundations of Engineering Design Languages (SFEDL)*, Grenoble, France, April 2002.   See p. 54, 96

[16] Gregory W. Bond, Franjo Ivancic, Nils Klarlund, and Richard Trefler. Eclipse feature logic analysis. In *2nd IP-Telephony Workshop*, pages 100–107, New York City, USA, April 2001.   See p. 73, 96

[17] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modelling Language: User Guide.* Addison-Wesley, 1999.   See p. 40, 43

[18] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.   See p. 46

[19] Charter—a tiny Java code generator for statecharts, September 2003. `http://www.mini.pw.edu.pl/~wasowski/projects/charter/`. See p. 48, 54

[20] Edmund M. Clarke. *Model Checking*. The MIT Press, December 1999.   See p. 6, 112

[21] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.   See p. 124

[22] Atmel Corporation. MARC4 microcontroller. `http://www.atmel.com/products/MARC4`. See p. 3

[23] Werner Damm, Bernhard Josko, Hardi Hungar, and Amir Pnueli. A compositional real-time semantics of STATEMATE designs. In Willem P. de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference, International Symposium (COMPOS)*, volume 1536 of *Lecture Notes in Computer Science*, pages 186–238, Bad Malente, Germany, September 1997. Springer-Verlag.   See p. 34

[24] Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, Dagstuhl Castle, Germany, February 1996. Springer-Verlag.   See p. 142

[25] Alexandre David. *Hierarchical Modeling and Analysis of Timed Systems*. PhD thesis, Uppsala Universitet, Department of Information Technology, Sweden, November 2003. IT Technical Report series 2003-050.   See p. 34

[26] Alexandre David, M. Oliver Möller, and Wang Yi. Formal verification of UML statecharts with real-time extensions. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering (FASE)*, volume 2306 of *Lecture Notes in Computer Science*, pages 218–232, Grenoble, France, April 2002. Springer-Verlag.   See p. 73, 96

[27] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120, Vienna, Austria, September 2001. ACM Press.   See p. 133

[28] Karsten Diethers, Ursula Goltz, and Michaela Huhn. Model checking UML statecharts with time. In Jürjens et al. [64], pages 35–51. TUM-I0208.

[29] Doron Drusinsky and David Harel. Using statecharts for hardware description and synthesis. *IEEE Transactions on Computer-Aided Design*, 8(7):798–807, 1989.

[30] Doron Drusinsky and David Harel. On the power of bounded concurrency I: Finite automata. *Journal of ACM*, 41(3):517–539, May 1994.

[31] Doron Drusinsky-Yoresh. A state assignment procedure for single-block implementation of state charts. *IEEE Transactions on Computer-Aided Design*, 10(12):1569–1576, 1991.

[32] Edwin Erpenbach. *Compilation, Worst-Case Execution Times and Schedulability Analysis of Statecharts Models*. PhD thesis, Department of Mathematics and Computer Science of the University of Paderborn, April 2000.

[33] Rik Eshuis and Roel Wieringa. Requirements level semantics for UML statecharts. In Scott F. Smith and Carolyn L. Talcott, editors, *Formal Methods for Open Object-Based Distributed Systems IV—Proc. FMOODS'2000, September, 2000, Stanford, California, USA*. Kluwer Academic Publishers, 2000.

[34] Sandro Etalle and Maurizo Gabbrieli. Partial evaluation of concurrent constraint languages. *ACM Computing Surveys*, 30(3es), September 1998.

[35] Elena Fersman, Paul Pettersson, and Wang Yi. Timed automata with asynchronous processes: Schedulability and decidability. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2280 of *Lecture Notes in Computer Science*, pages 67–82. Springer-Verlag, 2002.

[36] Ken Friis Larsen. *Types for DSP Assembler Programs*. PhD thesis, Technical University of Denmark (DTU) and IT University of Copenhagen (ITU), 2004.

[37] Ken Friis Larsen and Jakob Lichtenberg. MuDDy 2.0—SML interface to the binary decision diagrams package BuDDy. `http://www.itu.dk/research/muddy`. See p. 47, 52

[38] Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, and Jie Gong. *Specification and Design og Embedded Systems*. Prentice Hall, 1994. See p. 2

[39] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. See p. 40

[40] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. *IEEE Trans. Software Eng.*, 19(3):214–230, 1993. See p. 48

[41] Martin Gogolla and Cris Kobryn, editors. *4th International UML Conference—The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in Computer Science*, Toronto, Canada, October 2001. Springer-Verlag. See p. 210, 215

[42] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987. See p. 3, 20, 31, 33, 34, 96

[43] David Harel. Some thoughts on statecharts, 13 years later. In O. Grumberg, editor, *9th International Conference on Computer Aided Verification (CAV)*, volume 1254 of *Lecture Notes in Computer Science*, pages 226–231. Springer-Verlag, 1997. See p. 3

[44] David Harel and Hillel Kugler. The RHAPSODY semantics of statecharts (or, on the executable core of the UML). In Hartmut Ehrig, Werner Damm, Jörg Desel, Martin Große-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkämper, editors, *Integration of Software Specification Techniques for Applications in Engineering, Priority Program SoftSpez of the German Research Foundation (DFG), Final Report*, volume 3147 of *Lecture Notes in Computer Science*, pages 325–354. Springer-Verlag, 2004. See p. 35

[45] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996. See p. 33, 35, 73

[46] David Harel and Amir Pnueli. On the development of reactive systems. In Krzysztof R. Apt, editor, *Logic and Model of Concurrent Systems*, volume 13 of *NATO ASI*, pages 477–498. Springer-Verlag, October 1985. See p. 2

[47] David Harel, Amir Pnueli, Jeanette P. Schmidt, and Rivi Sherman. On the formal semantics of statecharts. In *2nd IEEE Symposium on Logic in Computer Science*, pages 54–64, New York, 1988. IEEE Computer Society Press.   See p. 20, 22, 34

[48] John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors. *Partial Evaluation: Practice and Theory. DIKU 1998 International Summer School*, volume 1706 of *Lecture Notes in Computer Science*. Springer-Verlag, Copenhagen, Denmark, 1999.   See p. 142

[49] Klaus Havelund and Grigore Roşu. Monitoring Java programs with Java PathExplorer. In *Proceedings of Workshop on Runtime Verification (RV'01)*, Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, 2001.   See p. 6

[50] Johannes Helbig and Peter Kelb. An OBDD-representation of statecharts. In Robert Werner, editor, *Proceeedings of The European Conference on Design Automation (EDAC)*, pages 142–149, Paris, France, Feb/Mar 1994. IEEE Computer Society Press.   See p. 46

[51] Thomas A. Henzinger, Christoph M. Kirsch, Marco A.A. Sanvido, and Wolfgang Pree. From control models to real-time code using Giotto. *IEEE Control Systems Magazine*, 23(1):50–64, February 2003.   See p. 54

[52] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, 1985.   See p. 115

[53] Gerard J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley, 2003.   See p. 45

[54] Haruo Hosoya, Naoki Kobayashi, and Akinori Yonezawa. Partial evaluation scheme for concurrent languages and its correctness. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par'96—Parallel Processing*, volume 1123 of *Lecture Notes in Computer Science*, pages 625–632, Lyon, France, August 1996. Springer-Verlag.   See p. 142

[55] C. Huizing and Willem P. de Roever. Introduction to design choices in the semantics of Statecharts. *Information Processing Letters*, 37:205–213, 1991.   See p. 17, 34

[56] C. Huizing, R. Gerth, and Willem P. de Roever. Modeling statecharts behavior in a fully abstract way. In Max Dauchet and Maurice Nivat, editors, *Proceedings of the 13th Colloquium on Trees in Algebra and Programming (CAAP)*, volume 299 of *Lecture Notes in Computer Science*, pages 271–294, Nancy, France, March 1988. Springer-Verlag.   See p. 34

[57] IAR Inc. IAR visualSTATE®.
`http://www.iar.com/Products/VS/`. See p. 10, 22, 73

[58] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the
pi-calculus. In *POPL 2001*. ACM Press, 2001. See p. 133

[59] International standard. Programming Languages—C. Ref. ISO/IEC
9899:1999(E). See p. 19, 39, 48

[60] Peter Jacobsen. Code generation for embedded systems. Master's
thesis, Technical University of Denmark (Lyngby) and IT University
of Copenhagen, April 1999. See p. 46, 95

[61] David N. Jansen. Probabilistic UML statecharts for specification and
verification: a case study. In Jürjens et al. [64]. TUM-I0208. See p.
34, 36

[62] Ralph E. Johnson and Jonathan Zweig. Delegation in C++. *Journal
of Object-Oriented Programming*, 4(11):22–35, November 1991. See
p. 40

[63] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial
Evaluation and Automatic Program Generation*. International Series
in Computer Science. Prentice Hall, 1993.
`http://www.dina.kvl.dk/~sestoft/pebook`. See p. 6, 142

[64] Jan Jürjens, Maria Victoria Cengarle, Eduardo B. Fernandez,
Bernhard Rumpe, and Robert Sandner, editors. *pUML Group
Workshop on Critical Systems Development with UML (CSDUML)*,
Dresden, Germany, September 2002. Technical University of Munich.
TUM-I0208. See p. 207, 210, 217

[65] Alexander Knapp and Stephan Merz. Model checking and code
generation for UML state machines and collaborations. In Dominik
Haneberg, Gerhard Schellhorn, and Wolfgang Reif, editors, *5th
Workshop on Tools for System Design and Verification*, Technical
Report 2002-11, pages 59–64. Institut für Informatik, Universität
Augsburg, 2002. See p. 95

[66] Donald E. Knuth. *The Art of Computer Programming*, volume 1.
Addison-Wesley, 3rd. edition, 1997. See p. 124

[67] Sabine Kuske. A formal semantics of UML state machines based on
structured graph transformation. In Gogolla and Kobryn [41], pages
241–256. See p. 34, 36

[68] Marcel Kyas, Harald Fecher, Frank S. de Boer, Joos Jacob, Jozef
Hooman, Mark van der Zwaag, Tamarah Arons, and Hillel Kugler.

Formalizing uml models and ocl constraints in pvs. In Mendler [91], pages 37–44. To be published in ENTCS.   See p. 36

[69] Kim G. Larsen. *Context Dependent Bisimulation Between Processes.* PhD thesis, Edinburgh University, 1986.   See p. 104, 131, 132, 133

[70] Kim G. Larsen. A context dependent equivalence between processes. *Theoretical Computer Science*, 49:184–215, 1987.   See p. 104, 132

[71] Kim G. Larsen, Ulrik Larsen, Brian Nielsen, Arne Skou, and Andrzej Wąsowski. Danfoss EKC trial project deliverables. Technical Report RS-03-48, BRICS, Aalborg, Denmark, December 2003.   See p. 3, 6, 49

[72] Kim G. Larsen, Ulrik Larsen, and Andrzej Wąsowski. Color-blind specifications for transformations of reactive synchronous programs. In Maura Cerioli, editor, *Proceedings of FASE, Edinburgh, UK, April 2005*, Lecture Notes in Computer Science. Springer-Verlag, 2005. Accepted.   See p. 132, 143

[73] Kim G. Larsen and Robin Milner. A compositional protocol verification using relativized bisimulation. *Information and Computation*, 99(1):80–108, 1992.   See p. 104, 132

[74] Hung Ledang. Automatic translation from UML specifications to B. In *16th IEEE International Conference on Automated Software Engineering (ASE)*, page 436nn, San Diego, CA, USA, November 2001. IEEE Computer Society Press.   See p. 34, 36

[75] Hung Ledang and Jeanine Souquières. Formalizing UML behavioral diagrams with B. In *Tenth OOPSLA Workshop on Behavioral Semantics: Back to Basics*, Tampa Bay, Florida, USA, October 2001. See p. 34, 36

[76] Hung Ledang and Jeanine Souquières. Contributions for modelling UML state-charts in B. In Michael J. Butler, Luigia Petre, and Kaisa Sere, editors, *Third International Conference on Integrated Formal Methods (IFM)*, volume 2335 of *Lecture Notes in Computer Science*, pages 109–127, Turku, Finland, 2002. Springer-Verlag.   See p. 34, 36

[77] Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.   See p. 33

[78] Johan Lilius and Iván Porres Paltor. Formalising UML state machines for model checking. In Robert B. France and Bernhard

Rumpe, editors, *The Unified Modeling Language—Beyond the Standard, Second International Conference (UML)*, volume 1723 of *Lecture Notes in Computer Science*, pages 430–445, Fort Collins, CO, USA, October 1999. Springer-Verlag.   See p. 34, 36

[79] Johan Lilius and Iván Porres Paltor. The semantics of UML state machines. Technical Report No 273, Turku Centre for Computer Science, Àbo Akademi University, Finland, May 1999.   See p. 34, 36

[80] Jørn Lind Nielsen. BuDDy – a binary decision diagram package version 2.0. http://www.it-c.dk/research/buddy.   See p. 47, 52

[81] Jørn Lind-Nielsen, Henrik R. Andersen, Henrik Hulgaard, Gerd Behrmann, Kåre J. Kristoffersen, and Kim G. Larsen. Verification of large state/event systems using compositionality and dependency analysis. *Formal Methods in System Design*, 18(1):5–23, January 2001.   See p. 142

[82] Jørn Bo Lind Nielsen. *Verification of Large/State Event Systems*. PhD thesis, Technical University of Denmark, April 2000.   See p. 35, 46, 142

[83] Gerard Lüttgen, Michael von der Beeck, and Rance Cleaveland. A compositional approach to statecharts semantics. In *ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pages 120–129, San Diego, California, USA, November 2000. ACM Press.   See p. 34

[84] Nancy Lynch. I/O automata: A model for discrete event systems. In *Annual Conference on Information Sciences and Systems*, pages 29–38, Princeton University, Princeton, N.J., 1988.   See p. 133

[85] Andrea Maggiolo-Schettini and Simone Tini. On disjunction of literals in triggers of statecharts transitions. *Information Processing Letters*, 84(6):305–310, October 2002.   See p. 34

[86] Florence Maraninchi. The Argos language: Graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, Kobe, Japan, October 1991.   See p. 33, 96

[87] Florence Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In Rance Cleaveland, editor, *Third International Conference on Concurrency Theory (CONCUR)*, volume 630 of *Lecture Notes in Computer Science*, pages 550–564, Stony Brook, NY, USA, August 1992. Springer-Verlag.   See p. 34

[88] Florence Maraninchi and Nicolas Halbwachs. Compiling ARGOS into boolean equations. In *Proc. 4th Int. School and Symposium on Formal Techniques in Real Time and Fault Tolerant Systems (FTRTFT)*, volume 1135 of *Lecture Notes in Computer Science*, Uppsala, Sweden, September 1996. Springer-Verlag. See p. 34, 46

[89] Florence Maraninchi and Yann Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, 27(1–3):61–92, 2001. See p. 33

[90] George H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955. See p. 71

[91] Michael Mendler, editor. *Workshop on Semantic Foundations of Engineering Design Languages (SFEDL)*. Elsevier Science Publishers, 2004. To be published in ENTCS. See p. 210, 215, 217

[92] Erich Mikk, Yassine Lakhnech, Carsta Petersohn, and Michael Siegel. On formal semantics of statecharts as supported by STATEMATE. In *2nd BCS-FACS Northern Formal Methods Workshop*. Springer-Verlag, 1997. See p. 34

[93] Erich Mikk, Yassine Lakhnech, and Michael Siegel. Hierarchical automata as model for statecharts. In R. K. Shyamasundar and Kazunori Ueda, editors, *Third Asian Computing Science Conference (ASIAN)*, volume 1345 of *Lecture Notes in Computer Science*, pages 181–196, Kathmandu, Nepal, December 1997. Springer-Verlag. See p. 34

[94] Robin Milner. *Communication and Concurrency*. Prentice Hall International Series in Computer Science. Prentice Hall, 1989. See p. 115, 132

[95] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997. See p. 47, 56

[96] E. F. Moore. Gedanken-experiments on sequential machines. Technical report, Automata studies, Princeton University, 1956. See p. 15

[97] Masaki Murakami. Partial evaluation of reactive communciating processes using temporal logic formulas. In *Workshop on Algebraic and Object-Oriented Approaches to Software Science*, 1995. See p. 142

[98] Object Management Group. OMG Unified Modelling Language specification, 1999. `http://www.omg.org`. See p. 6, 31, 47, 124

[99] Object Management Group. Model driven architecture (MDA)—a technical perspective, 2001. http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01. See p. 6, 133

[100] Object Management Group. Meta object facility (MOF) specification, 2002. http://www.omg.org. See p. 47

[101] Object Management Group. OMG XML metadata interchange (XMI) specification, January 2002. http://www.omg.org. See p. 47

[102] Object Management Group (OMG). UML 2.0 infrastructure specification, December 2003. Final Adopted Specification (in finalization phase) http://www.omg.org, document signature ptc/03-09-15.pdf. See p. 34, 35

[103] Object Management Group (OMG). UML 2.0 superstructure specification, August 2003. Final Adopted Specification (in finalization phase) http://www.omg.org, document signature ptc/03-08-02.pdf. See p. 22, 27, 31, 35, 36, 132

[104] Karsten Andreas Pihl, Mette Berger, and Lone Gram Larsen. Code generation for the mini-statechart language. 4-weeks project report, IT University of Copenhagen, May 2003. See p. 48, 54

[105] Gergely Pintér and István Majzik. Automatic implementation of extended hierarchical automata. Technical report, Budapest University of Technology and Economics, Budapest, 2003. See p. 45

[106] Gergely Pinter and Istvan Majzik. Impact of statechart implementation techniques on the effectiveness of fault detection mechanisms. In *Proceedings of the 30th EUROMICRO Conference*. IEEE Computer Society Press, 2004. See p. 43, 45, 54

[107] Kristofer S. J. Pister, J. M. Kahn, and B. E. Boser. Smart dust: Wireless networks of millimeter-scale sensor nodes. Highlight Article in 1999 Electronics Research Laboratory Research Summary, 1999. See p. 2

[108] Amir Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In Takayasu Ito and Albert R. Meyer, editors, *International Conference on Theoretical Aspects of Computer Software (TACS)*, volume 526 of *Lecture Notes in Computer Science*, pages 244–264, Sendai, Japan, September September 24-27, 1991. Springer-Verlag. See p. 34, 96

[109] Sriram K. Rajamani and Jakob Rehof. Conformance checking for models of asynchronous message passing software. In Ed Brinksma and Kim Guldstrand Larsen, editors, *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 166–179, Copenhagen, Denmark, July 2002. Springer-Verlag. See p. 133

[110] S. Ramesh. Efficient translation of statecharts into hardware circuits. In *12th International Conference on VLSI Design*, pages 384–389. IEEE Computer Society Press, January 1999. See p. 95

[111] Arnab Ray, Rance Cleaveland, and Arne Skou. An algebraic theory of boundary crossing transitions. In Mendler [91], pages 63–81. To be published in ENTCS. See p. 34

[112] Matthias Riebisch, Ilka Philippow, and Marco Götze. UML-based statistical test case generation. In Mehmet Aksit, Mira Mezini, and Rainer Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World. NetObjectDays, Revised Papers*, volume 2591 of *Lecture Notes in Computer Science*, pages 394–411, Erfurt, Germany, October 2002. Springer-Verlag. See p. 96

[113] Ella E. Roubtsova, Jan van Katwijk, Ruud C. M. de Rooij, and Hans Toetenel. Transformation of UML specification to XTG. In Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin, editors, *Perspectives of System Informatics (PSI), 4th International Andrei Ershov Memorial Conference, Revised Papers*, volume 2244 of *Lecture Notes in Computer Science*, pages 249–256, Akademgorodok, Novosibirsk, July 2001. Springer-Verlag. See p. 73

[114] Miro Samek. *Practical Statecharts in C/C++*. CMP Books, Lawrence, Kansas, 2002. See p. 43, 45, 54, 147

[115] Helmut Seidl, Varmo Vene, and Markus Müller-Olm. Towards a practical analyzer for multi-threaded C. In *14th Nordic Workshop on Programming Theory (NWPT). Abstracts*, pages 90–92, Tallin, Estonia, nov 2002. Institute of Cybernetics and Tallin Technical University. See p. 6

[116] Emil Sekerinski and Rafik Zurob. iState: A statechart translator. In Gogolla and Kobryn [41], pages 376–390. See p. 95

[117] Bran Selic. An efficient object-oriented variation of the statecharts formalism for distributed real-time systems. In David Agnew, Luc J. M. Claesen, and Raul Camposano, editors, *Proceedings of the 11th IFIP WG10.2 International Conference on Computer Hardware*

Description Languages and their Applications, volume A-32 of *IFIP Transactions*, pages 335–344, Ottawa, Ontario, Canada, April 1993. IFIP.   See p. 33, 35

[118] Anthony J. H. Simons. The compositional properties of UML statechart diagrams. In C. J. van Rijsbergen, editor, *Third Electronic Workshop on Rigorous Object-Oriented Methods*. British Computer Society, 2000.   See p. 36, 73

[119] Jørgen Staunstrup, Henrik Reif Andersen, Henrik Hulgaard, Jørn Lind-Nielsen, Kim Guldstrand Larsen, Gerd Behrmann, Kaare J. Kristoffersen, Arne Skou, Henrik Leerberg, and Niels Bo Theilgaard. Practical verification of embedded software. *IEEE Computer*, 5(33):68–75, 2000.   See p. 73

[120] Jørgen Steensgaard-Madsen. Dulce: Danish uniform language composition engine. `http://www.imm.dtu.dk/~jsm/dulce`.   See p. 54

[121] Jørgen Steensgaard-Madsen. Htel: a hypertext expression language. *Software—Practice and Experience*, 29(8):661–675, 1999.   See p. 54

[122] Sun Microsystems, Inc. Java card(TM) specification. `http://java.sun.com/products/javacard/specs.html`.   See p. 124

[123] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.   See p. 101

[124] Issa Traoré. An outline of PVS semantics for UML statecharts. *Journal of Universal Computer Science*, 6(11):1088–1108, 2000.   See p. 34, 36

[125] Jan Tretmans. Testing concurrent systems: A formal approach. In Jos C. M. Baeten and Sjouke Mauw, editors, *CONCUR '99: Concurrency Theory, 10th International Conference, Eindhoven, The Netherlands, August 24–27, 1999, Proceedings*, volume 1664 of *Lecture Notes in Computer Science*, pages 46–65. Springer-Verlag, 1999.   See p. 6

[126] Rob van Glabbeek. The linear time–branching time spectrum (extended abstract). In J.C.M. Beaten and J.W. Klop, editors, *Theories of Concurrency: Unification and Extension (CONCUR)*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297, Amsterdam, The Netherlands, August 1990. Springer-Verlag.   See p. 37

[127] Rob van Glabbeek. The linear time—branching time spectrum I. In Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors, *Handbook of Process Algebra*, pages 3–99. Elsevier Science Publishers, North-Holland, 2001. See p. 37

[128] Michael von der Beeck. A comparison of statecharts variants. In *Third International Symposium on Formal Techniques in Real Time and Fault-Tolerant Systems (FTRTFT)*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148, Lübeck, Germany, September 1994. Springer-Verlag. See p. 34, 53

[129] Yunming Wang, Jean-Pierre Talpin, Albert Benveniste, and Paul Le Guernic. A semantics of UML state-machines using synchronous pre-order transition systems. In *Proceedings of Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 96–103, Newport Beach, CA, USA, March 2000. IEEE Computer Society Press. See p. 34, 36

[130] Andrzej Wąsowski. On Efficient Program Synthesis from Statecharts. In *ACM SIGPLAN Languages, Compilers, and Tools for Embedded Systems (LCTES)*, San Diego, USA, June 2003. ACM Press. See p. 94, 96

[131] Andrzej Wąsowski. Automatic generation of program families by model restrictions. In *Software Product Line Conference (SPLC)*, Lecture Notes in Computer Science, Boston, USA, August/September 2004. Springer-Verlag. See p. 132, 143

[132] Andrzej Wąsowski. Flattening Statecharts without Explosions. In *ACM SIGPLAN Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 257–266, Washington DC, USA, June 2004. ACM Press. See p. 94

[133] Andrzej Wąsowski. On Succinctness of Hierarchical State Diagrams in Absence of Message Passing. In Mendler [91]. To be published in ENTCS. See p. 94

[134] Andrzej Wąsowski. Succinctness of hierarchical state diagrams in absence of message passing. Technical Report 42, IT University of Copenhagen, Denmark, February 2004. See p. 94

[135] Andrzej Wąsowski and Peter Sestoft. Compile-time scope resolution for statecharts transitions. In Jürjens et al. [64], pages 133–145. TUM-I0208. See p. 54

[136] Andrzej Wąsowski and Peter Sestoft. On the formal semantics of visualSTATE statecharts. Technical Report TR-2002-19, IT University of Copenhagen, September 2002. See p. 34

[137] Andrzej Wąsowski. SCOPE: A statechart compiler, 2002-2004.
      `http://www.mini.pw.edu.pl/~wasowski/scope`. See p. 22, 45, 56

[138] Albert Zündorf. Rigorous object oriented software development with
      Fujaba v. 0.3. Unpublished Draft, 2002. See p. 43, 45, 54, 95

Index