Modeling software product lines using color-blind transition systems

Larsen, Kim Guldstrand; Nyman, Ulrik; Wasowski, Andrzej

Published in:
International Journal on Software Tools for Technology Transfer

DOI (link to publication from Publisher):
10.1007/s10009-007-0046-x

Publication date:
2007

Document Version
Publisher's PDF, also known as Version of record

Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Modeling Software Product Lines Using Color-blind Transition Systems

Kim G. Larsen[1], **Ulrik Nyman**[1], **Andrzej Wąsowski**[2]

[1] CISS, Aalborg University, Denmark, e-mail: {kgl,ulrik}@cs.aau.dk
[2] IT University of Copenhagen, Denmark, e-mail: wasowski@itu.dk

Received: date / Revised version: date

Abstract. Families of embedded discrete finite state programs are modeled using input-enabled alternating transition systems. One model describes all functionality, while each variant is defined by an environment, describing its possible uses. The environments show both the inputs that a system can receive and indicate which of the system's responses are relevant for the environment. The latter trait, called color-blindness, creates new possibilities for system transformations in the specialization process. We demonstrate the use of the framework by applying it to two classes of realistic design languages. An example of a product line of alarm clocks is used throughout the article.

Key words: Product Lines, Embedded Software, Labeled Transition Systems, Modeling, Relativized Simulation

1 Introduction

Modern software becomes increasingly customizable. Embedded devices are often produced in multiple variants, each needing different software. Our long-term goal is to provide a theoretical foundation, tools, and a methodology for maintenance of a family of embedded software products with similar but varying degree of functionality. Such a family of products is usually known as a product line, and the process of maintaining the software is known as product line management [5,11]. The present work focuses on a framework for modeling software product lines and specifying correctness of transformations used in automatic derivation of family members.

In design of embedded software the final code size is often essential for the financial success of the product. Even a small reduction in code size may allow reductions in the production cost, if it makes it possible to use cheaper hardware. Our communications with vendors confirm that many electronic systems are produced in such large quantities that even a small reduction for each device can bring enormous savings.

Manual implementation of the different versions of software for different versions of the product can keep the final code sizes close to optimal, while driving up the software development cost significantly. The general idea of product line management is to avoid explicit maintenance of multiple versions of the software, and use methods for design and development of the entire product family as a whole.

We propose to use a single general model as a description of all available functionality in a product family. Such a family may evolve over time, and so can the general model, but we do not consider this here. A set of hierarchically organized specifications describes the different environments in which each version of the embedded software will operate. Pragmatically speaking environments are descriptions of all possible uses for a given variant of the product. A simpler system variant usually allows less sensor inputs and less actuator outputs, which corresponds to environments *not providing* the inputs, or *not caring* about specific responses received. We will say that the implementation of the system has been specialized correctly to a given variant if the restricting environment cannot see the difference between the original and specialized versions of the program.

Fig. 1 gives an overview of the setup. The designer needs to create the model containing the complete set of functionality (top-left state diagram) and the environment specifications (top row). In the depicted setup the general model is specialized to a given environment in the model language before being compiled into the target language. The model obtained after the specialization should behave identically to the general model as

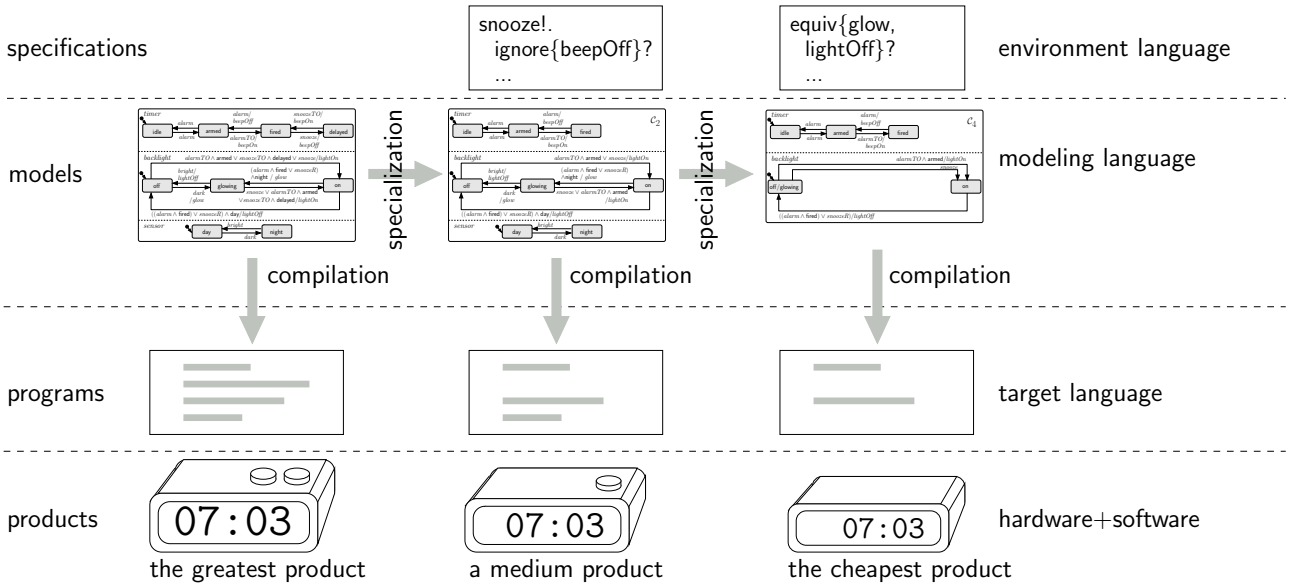

Fig. 1. Overview of a product line of alarm clocks. The top left model is the general model specified by a developer. User also provides usage descriptions in the form of environments (on the very top).

long as the environment behaves according to its specification. In pragmatic terms this means that if the environment models the hardware and the system is specialized with respect to this model, it should behave as expected as long as it is run on the hardware respecting the assumptions.

A compelling example of how different two versions of the same product can be, can be found on Fig. 2, which shows the front of two different coffee vending machines by Wittenborg. The simple machine, FB 55, has no coin collection unit or choice of additives, such as sugar or milk. Such a simple coffee machine is ideal in circumstances where the coffee is pre-paid, and where the simplicity is important because people are in a hurry. Indeed many even simpler Wittenborg's machines have recently been seen at open areas of Frankfurt and Munich airports. The more complex machine, ES 5100, is more appropriate for an environment where people use the same machine every day, such as at work (one can be seen in CISS, where two of the authors are employed). The control software for these two machines could have been constructed by creating one model containing the comprehensive functionality of the most complex coffee machine and then creating environments for simpler coffee machines. The environments could state that the user would never use certain buttons, in that these buttons would not be physically present on the machine.

Throughout the article we will use an alarm clock example. In order to produce several different versions of alarm clocks we only need to design a general model with all the functionality (Fig. 3) and specify the environments in which the other versions of the alarm clocks are expected to function. Fig. 7 shows a very restricted version of the alarm clock, with much less functionality.

Fig. 2. Two variants of Wittenborg's coffee machine. The simple FB 55 (left) has few buttons and no coin collection unit. The advanced ES 5100 (right) provides the user with the choice of different coffee specialties and additives.

This alarm clock can be derived from the original model given a description of an environment.

Our environments not only restrict possible input stimuli, but also exhibit inabilities in distinguishing output stimuli. Some outputs are indistinguishable for a given environment in the same way as a color-blind person cannot distinguish some colors. In the case of Fig. 7 the environment cannot, among other things, see the difference between the light in the clock glowing or being completely off. Inability of a given environment to distin-

guish certain outputs is dependent on the state of that environment. Thus an environment can specify that a user, after pressing one button, will not distinguish between two different outputs, but will distinguish between these two outputs again after pressing another button, e.g. a reset button.

The paper proceeds as follows. Section 2 introduces state/event systems and motivates our work using the alarm clock example. I/O alternating transition systems are presented in section 3, while the concept of color-blindness is introduced in section 4. Composition operators for environments are discussed in section 5. Section 6 focuses on adaptation to realistic design languages. The main alarm clock example is completed in section 7. In section 8 we briefly touch on available techniques that can be used in specializing code for product variants. Sections 9–10 refer the related work and conclude.

2 State/Event Systems

Let *Event* and *Action* be finite sets of environment stimuli and system outputs respectively. A *state/event machine* $M_i = (S_i, s_i^0, T_i)$ is a triple comprising a set of local states S_i, the initial state $s_i^0 \in S_i$ and a set of syntactic transitions T_i. A *state/event system* consists of n machines $\mathcal{M} = \{M_1, \ldots, M_n\}$ with mutually disjoint sets of states. A global state of the system is a tuple of local states: $State = S_1 \times S_2 \times \cdots \times S_n$. Transitions in $T_i \subseteq S_i \times Event \times Guard \times Action \times S_i$ describe reactions undertaken by M_i in reply to a given event, in a given local and global state. Global states are described by transition guards: simple Boolean expressions over activity of states, which can be evaluated in any given global state, giving rise to a natural satisfaction relation $\models \subseteq State \times Guard$.

State/event systems are *input-enabled*: the local transition relation includes not only the syntactical transitions but also self loops for all configurations for which reactions are not specified. We write $s \xrightarrow{e \quad o}_i s'_i$, meaning that the reaction of machine M_i to arrival of event e in global state s is, to change the local state to s'_i and generate the set of actions o:

$$\begin{array}{ll} s \xrightarrow{e \quad \{a\}}_i s'_i & \text{iff } \exists g.\ (\pi_i(s), e, g, a, s'_i) \in T_i \wedge s \models g \\ s \xrightarrow{e \quad \emptyset}_i \pi_i(s) & \text{otherwise,} \end{array}$$

where $\pi_i(s)$ denotes the i'th projection of s. The global transition relation

$$T \subseteq State \times Event \times \mathcal{P}(Action) \times State$$

subsumes all local reactions

$$s \xrightarrow{e \quad o} s' \Leftrightarrow_{def} \forall i. s \xrightarrow{e \quad o_i}_i \pi_i(s') \text{ where } o = o_1 \cup \ldots \cup o_n.$$

Fig. 3 depicts a state/event model $\mathcal{C}_0$ of an alarm clock. The essentials 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. The actual timers are not part of the modeled system, and thus the environment sends a time-out event to the system with a certain delay after a hardware timer has been activated. The user can postpone the alarm by pressing the snooze button (event *snooze*), which allows him to continue sleeping until the snooze timer times out (*snoozeTO*). Releasing the button sends a *snoozeR* event to the model. The *backlight* machine controls the built-in lamps. Only a faint light is displayed in the glowing state, such that the display can be read in the dark. The full light is on while the alarm is beeping or the snooze button is being pressed. The *sensor* machine models the current external light level. Events *dark* and *bright* are generated by the sensor driver whenever the light level around the clock changes above or below some threshold.

We would like to support automatic derivation of model variants for discrete control systems like the alarm clock. One such variant $\mathcal{C}_4$, which is a very limited version of the alarm clock is depicted on Fig. 7. This variant does not have any snooze functionality, because the environment guarantees that it will never provide the input *snooze*. Thus it is as if there were no snooze button on the alarm clock. This environment also specifies that it cannot see the difference between the backlight glowing and being completely off. This implies that the hardware of the clock would only need one lightbulb, for the full light setting, instead of two lightbulbs for two different light levels. The glowing and off states can now be combined into one state and the *sensor* component is no longer necessary.

3 I/O Alternating Transition Systems

The reactive synchronous paradigm seems to be predominant in development of embedded software. The state/event systems of the previous section [24,16] are just an example chosen from a multitude of available formalisms, like Esterel [2], statecharts [12], or Java Card [32]. A common assumption about these systems is that they react to any input event at any time and each reaction occurs infinitely fast, so that the system is always able to observe the arrival of the next event. Such semantics is conveniently captured by *I/O-alternating transition systems*:

Definition 1. An I/O-alternating transition system, or IOATS, is a tuple $(In, Out, Gen, Obs, \xrightarrow{!}, \xrightarrow{?}, s^0)$, where In and Out are sets of inputs and outputs, Gen and Obs are finite sets of generators and observers, $\xrightarrow{!} \subseteq Gen \times Out \times Obs$ is a generation relation, $\xrightarrow{?} \subseteq Obs \times In \times Gen$ is an observation relation, and $s^0 \in Gen \cup Obs$ is the initial state.

We have distinguished two transition relations: $\xrightarrow{!}$ is a generation relation advancing from a generator to an

observer, while $\xrightarrow{?}$ is an observation relation advancing from an observer to a generator. This alternation is inherent to the way synchronous systems operate. We write $S \xrightarrow{o!} s$, instead of $(S, o, s) \in \xrightarrow{!}$ and $s \xrightarrow{i?} S$ instead of $(s, i, S) \in \xrightarrow{?}$. Small letters are used for observers and capital letters for generators. In addition observers are required to be input-enabled:

$$\forall s \in Obs.\ \forall i \in In.\ \exists S, o, s'.\ s \xrightarrow{i?} S \wedge S \xrightarrow{o!} s' \qquad (1)$$

With these assumptions we can propose a simulation based refinement relation:

Definition 2. 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 IOATSs. A binary relation $R \in Obs_1 \times Obs_2$ constitutes a simulation on observers of $\mathcal{S}_1$ and $\mathcal{S}_2$ iff $(s_1, s_2) \in R$ implies that:

whenever $s_1 \xrightarrow{i?} S_1 \wedge S_1 \xrightarrow{o!} s_1'$
then also $s_2 \xrightarrow{i?} S_2 \wedge S_2 \xrightarrow{o!} s_2'$ and $(s_1', s_2') \in R$.

Let R be the largest of such relations ordered by inclusion. An observer s_2 simulates an observer s_1, written $s_1 \leqslant s_2$, iff $(s_1, s_2) \in R$. Finally $\mathcal{S}_2$ simulates $\mathcal{S}_1$, written $\mathcal{S}_1 \leqslant \mathcal{S}_2$, iff $s_1^0 \leqslant s_2^0$.

We distinguish the actual systems from the environments, in which they operate. Environments are free in choice of inputs, while systems independently determine the outputs. 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 $s_{\mathcal{S}} \in Obs_{\mathcal{S}}$. Environments always begin execution in a generator state, so $s_{\mathcal{E}} \in Gen_{\mathcal{E}}$. System $\mathcal{S}$ is *compatible* with the environment $\mathcal{E}$ if $In_{\mathcal{S}} = Out_{\mathcal{E}}$ and $Out_{\mathcal{S}} = In_{\mathcal{E}}$. Composition of a system $\mathcal{S}$ with a compatible environment $\mathcal{E}$ is defined in the usual way, by synchronization on identical labels (and complementary transition types). 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. For a closed system it is known, which of its states cannot 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 with a notion of *relativized simulation*:

Definition 3. 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 $\mathcal{S}_2 = (In, Out, Gen_2, Obs_2, \xrightarrow{!}_2, \xrightarrow{?}_2, s_2^0)$. A Gen-indexed family of binary relations $R: Gen \to \mathcal{P}(Obs_1 \times Obs_2)$ is a relativized simulation iff $(s_1, s_2) \in R_E$ implies that:

whenever $E \xrightarrow{i!} e \wedge e \xrightarrow{o?} E'$
then whenever $s_1 \xrightarrow{i?} S_1 \wedge S_1 \xrightarrow{o!} s_1'$
then also $s_2 \xrightarrow{i?} S_2 \wedge S_2 \xrightarrow{o!} s_2'$ and $(s_1', s_2') \in R_{E'}$.

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

The choice of simulation as the preorder underlying our methodology is somewhat arbitrary. Most other behavioral preorders of the linear-time/branching-time hierarchy of van Glabbeek [35] would be adequate, such as testing preorder, ²/₃ bisimulation (ready simulation), language inclusion, ioco [34] and bisimulation. What is important is that the particular preorder preserves properties of interest and that the preorder may be relativized with respect to environmental restrictions.

4 Color-blind I/O-alternating Transition Systems

In the previous section we have stated that two systems are in a refinement relation 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 *different outputs* might be limited depending on its actual state. A variant of the alarm clock may have only one lightbulb installed, which should be lit whenever the backlight is on or glowing. The environment, being a model of the hardware in this case, will treat the *glow* and *lightOn* outputs as identical, allowing for optimizations when generating code for this specific type of hardware.[1] 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 relabelling and hiding. However, environmental restrictions can be dynamically changing. This is the case for the environment leading to the specialized model $\mathcal{C}_1$ (Fig. 4). Here the environment only becomes blind for the *lightOn* action after the production of the *snooze* event. To give a proper treatment of such situations we relax the equivalence of labels in relativized simulation and label observation transitions of environments with sets of inputs called *observation classes*. Such transitions can be taken in the presence of any of the inputs in their observation class.

Definition 4. A *color-blind* IOATS is a tuple $\mathcal{E} = (In, Out, Gen, Obs, \xrightarrow{!}, \xrightarrow{?}, E^0)$, where In and Out are sets of inputs and outputs, Gen and Obs are finite sets of generators and color-blind observers, $\xrightarrow{!} \subseteq Gen \times Out \times Obs$ is a generation relation, $\xrightarrow{?} \subseteq Obs \times \mathcal{P}(In) \times Gen$ is a color-blind observation relation, and $E^0 \in Gen$ is an initial state.

[1] Note that this is an alternative to the version presented in Fig. 7, where *glow* and *lightOff* are considered identical.

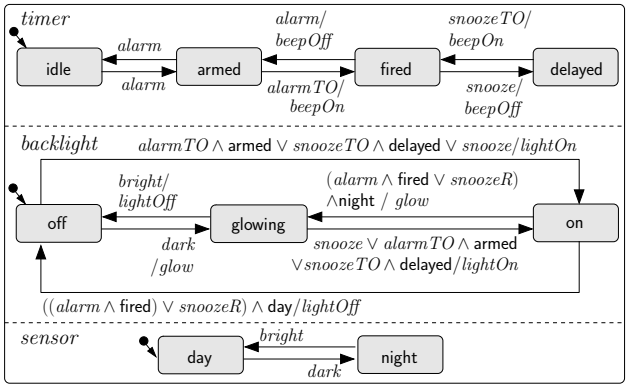

Fig. 3. The initial model, $\mathcal{C}_0$, of the alarm clock

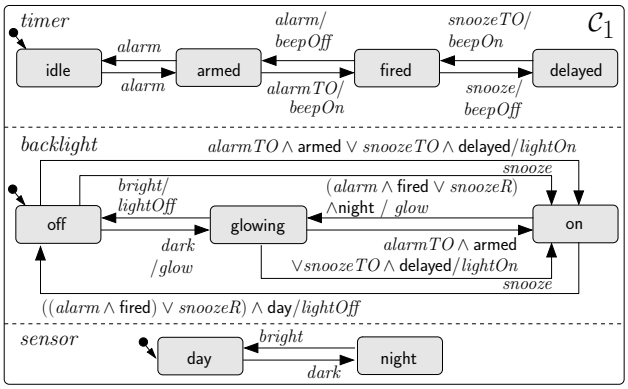

Fig. 4. A specialized model, $\mathcal{C}_1$, of an alarm clock.

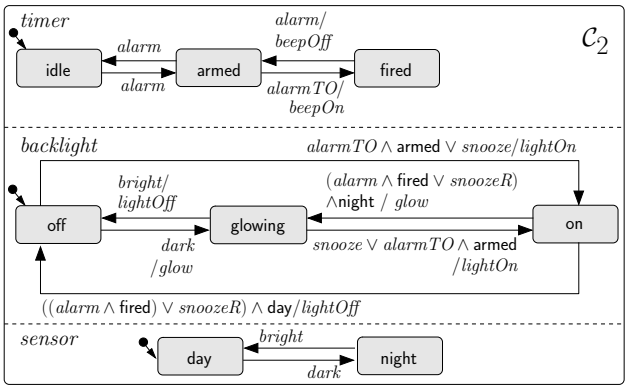

Fig. 5. A specialized model, $\mathcal{C}_2$, of an alarm clock.

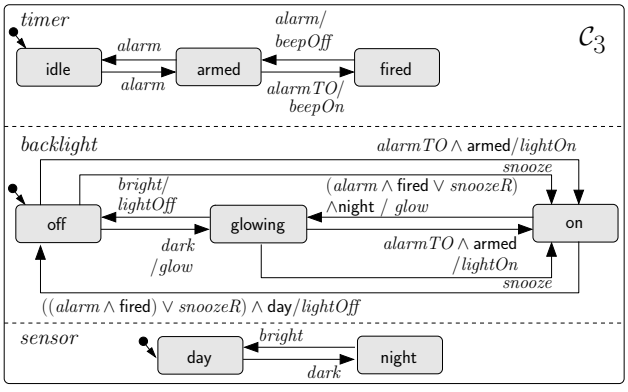

Fig. 6. A specialized model, $\mathcal{C}_3$, of an alarm clock.

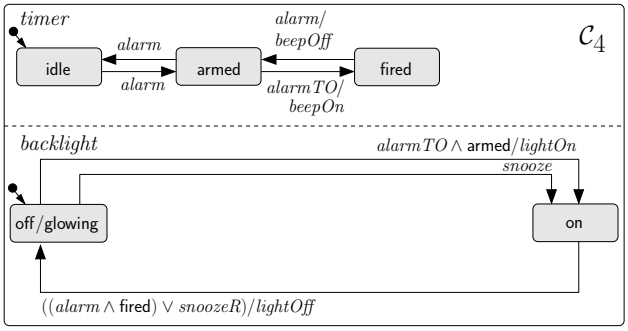

Fig. 7. A specialized model, $\mathcal{C}_4$, of an alarm clock

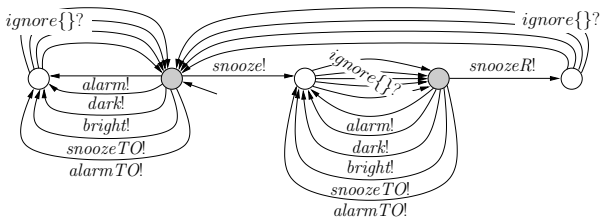

Fig. 8. Environment *Interleave snooze snoozeR*.

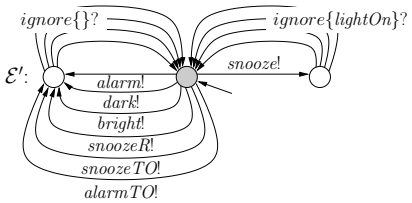

Fig. 9. Environment $\mathcal{E}'$ ignoring the *lightOn* output produced in reaction to the *snooze* button.

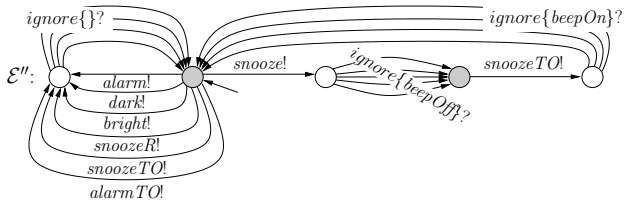

Fig. 10. Environment $\mathcal{E}''$ ignoring the snooze function of the clock.

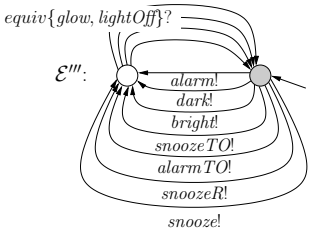

Fig. 11. Environment $\mathcal{E}'''$ *Equiv glow lightOff*.

A color-blind environment $\mathcal{E} = (In_\mathcal{E}, Out_\mathcal{E}, Gen_\mathcal{E}, Obs_\mathcal{E}, \xrightarrow{!}_\mathcal{E}, \xrightarrow{?}_\mathcal{E}, E)$ and a usual IOATS $\mathcal{S} = (In_\mathcal{S}, Out_\mathcal{S}, Gen_\mathcal{S}, Obs_\mathcal{S}, \xrightarrow{!}_\mathcal{S}, \xrightarrow{?}_\mathcal{S}, s)$ are compatible if their signatures match: $In_\mathcal{E} = Out_\mathcal{S} \wedge Out_\mathcal{E} = In_\mathcal{S}$. Since we only consider compatible systems and environments, we fix the meaning of the input and output, choosing the system's perspective. We denote the set of inputs of the system by In (which is also 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). A single input will be denoted by i, single output by o, and classes of outputs by capital O. We still write $E \xrightarrow{i!} e$ instead of $(E, i, e) \in \xrightarrow{!}$ and $e \xrightarrow{O?} E$ instead of $(e, O, E) \in \xrightarrow{?}$.

We require that the observers in color-blind IOATS are deterministic and input enabled, so that the observation classes on the transitions outgoing from a single state form a partitioning of the inputs into equivalence classes. Formally:

$$\begin{aligned}
&\forall e \in Obs_\mathcal{E}.\forall O_1, O_2 \subseteq Out.\forall E_1, E_2 \in Gen_\mathcal{E}.\\
&\quad e \xrightarrow{O_1?} E_1 \wedge e \xrightarrow{O_2?} E_2\\
&\quad \Rightarrow O_1 \cap O_2 = \emptyset \vee (O_1 = O_2 \wedge E_1 = E_2)\\
&\forall e \in Obs_\mathcal{E}.\forall o \in Out.\exists O \subseteq Out.\exists E \in Gen_\mathcal{E}.\\
&\quad e \xrightarrow{O?} E \wedge o \in O. \qquad (2)
\end{aligned}$$

The generation relation should also be deterministic:

$$\forall E \in Gen_\mathcal{E}.\ \forall i \in In.\forall e_1, e_2 \in Obs_\mathcal{E}.\\ E \xrightarrow{i!} e_1 \wedge E \xrightarrow{i!} e_2 \Rightarrow e_1 = e_2\ . \quad (3)$$

Note that determinism in this sense does not limit the freedom of the environment in choosing inputs, but means that each input choice uniquely determines the target state.

Consider a blind environment $\mathcal{B}$ with two states, a generator $\mathbf{B}$ and an observer $\mathbf{b}$. Intuitively $\mathcal{B}$ can execute all parts of the system, but does not care about the responses it gets:

$$\forall i \in In.\ \mathbf{B} \xrightarrow{i!} \mathbf{b} \text{ and } \mathbf{b} \xrightarrow{Out?} \mathbf{B}\ .$$

Dually, a perfect vision environment $\mathcal{V}$ observes all the outputs:

$$\forall i \in In.\mathbf{V} \xrightarrow{i!} \mathbf{v} \text{ and } \forall o \in Out.\mathbf{v} \xrightarrow{\{o\}?} \mathbf{V}\ .$$

A compatible environment–system pair forms a closed system, advancing in lock-steps. The generation transition of the system, synchronizes with the observation transition of the environment, whenever the output produced falls into the right observation class. We enrich our previous definition of relativized simulation to accommodate this new synchronization principle:

Definition 5. Let $\mathcal{E} = (Out, In, Gen, Obs, \xrightarrow{!}, \xrightarrow{?}, E^0)$ be a color-blind environment IOATS and $\mathcal{S}_1 = (In, Out, Gen_1, Obs_1, \xrightarrow{!}_1, \xrightarrow{?}_1, s_1^0)$, $\mathcal{S}_2 = (In, Out, Gen_2, Obs_2, \xrightarrow{!}_2, \xrightarrow{?}_2, s_2^0)$ be two system IOATSs. A Gen-indexed family of relations $R{:}Gen \to \mathcal{P}(Obs_1 \times Obs_2)$ is a relativized simulation iff $(s_1, s_2) \in R_E$ implies that:

$$\begin{aligned}
&\text{whenever } E \xrightarrow{i!} e \wedge e \xrightarrow{O?} E'\\
&\quad \text{then whenever } s_1 \xrightarrow{i?} S_1 \wedge S_1 \xrightarrow{o_1!} s_1' \wedge o_1 \in O\\
&\quad\quad \text{then also } s_2 \xrightarrow{i?} S_2 \wedge S_2 \xrightarrow{o_2!} s_2' \wedge o_2 \in O\\
&\quad\quad\quad \text{and } (s_1', s_2') \in R_{E'}.
\end{aligned}$$

Let R be the largest of such families ordered by component-wise inclusion. An observer s_2 simulates an observer s_1 in the context of generator E, written $s_1 \leqslant_E s_2$, iff $(s_1, s_2) \in R_E$. 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 $s_1^0 \leqslant_{E^0} s_2^0$. Finally $\mathcal{S}_1$ is equivalent to $\mathcal{S}_2$ in the context of $\mathcal{E}$, written $\mathcal{S}_1 \lessgtr_\mathcal{E} \mathcal{S}_2$, iff $\mathcal{S}_1 \leqslant_\mathcal{E} \mathcal{S}_2$ and $\mathcal{S}_2 \leqslant_\mathcal{E} \mathcal{S}_1$.

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 $\mathbb{R}$ be an endofunction on a Gen-indexed family of binary relations:

$$\begin{aligned}
&\mathbb{R}(R) = \lambda E.\{(s_1, s_2) \mid \forall i, e, O, E'.\ \forall i, S_1, o_1, s_1'.\ \exists S_2, o_2.\\
&\quad E \xrightarrow{i!} e \wedge e \xrightarrow{O?} E' \wedge s_1 \xrightarrow{i?} S_1 \wedge S_1 \xrightarrow{o_1!} s_1' \wedge o_1 \in O\\
&\qquad \text{implies } s_2 \xrightarrow{i?} S_2 \wedge S_2 \xrightarrow{o_2!} s_2' \wedge o_2 \in O \wedge (s_1', s_2') \in R_{E'}\}\ .
\end{aligned}$$

Proposition 1. *A Gen-indexed family of relations R constitutes a relativized simulation with respect to a color-blind IOATS iff $R \subseteq \mathbb{R}(R)$ (inclusion interpreted component-wise).*

Proof. $\mathbb{R}$ is a monotonic endofunction on the complete lattice of Gen-indexed famillies of binary relations over $Obs_1 \times Obs_2$ ordered by inclusion. By Tarski's theorem [33] $\mathbb{R}$ has the greatest fixpoint $\bigcap_{j=0}^{\infty} \mathbb{R}^j(\lambda E.\, Obs_{\mathcal{S}_1} \times Obs_{\mathcal{S}_2})$, equal to the relativized simulation of Def. 5. Since the fixpoint contains all relativized simulations, we can use a classic proof technique: to show that one IOATS simulates another in an environment, find any relativized simulation relating them. □

Even though we have initially postulated that typical execution contexts do not exercise all possible traces of the system, we shall now require that environments can always produce any of the inputs in In. This requirement surprisingly does not defeat our initial goal. We can direct all transitions producing impossible inputs to the observer $\mathbf{b}$ and embed the blind environment $\mathcal{B}$ with a suitable signature in every environment. Instead of specifying that the environment cannot produce i, we state that i can be produced, but the subsequent system behavior is irrelevant. Proposition 2 justifies this formally:

Proposition 2. *For any two observers s_1, s_2 from IOATSs $\mathcal{S}_1$, $\mathcal{S}_2$ with identical signatures: $s_1 \leqslant_\mathbf{B} s_2$ (where $\mathbf{B} \in \mathcal{B}$ such that $\mathcal{B}$ closes $\mathcal{S}_1$ and $\mathcal{S}_2$).*

Fig. 12 presents two systems and two compatible color-blind environments. Environment transitions from generators to the blind observer $\mathbf{b}$ have been omitted. There is one such transition for each input–generator pair, for which the transition is not drawn. Observe that

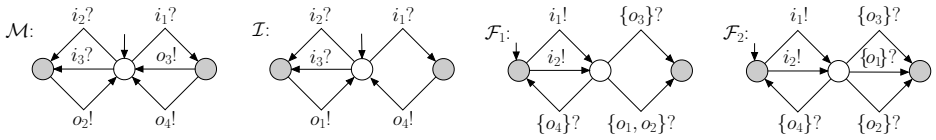

Fig. 12. Systems $\mathcal{M}$ and $\mathcal{I}$ and compatible environments $\mathcal{F}_1$, $\mathcal{F}_2$

the system $\mathcal{M}$ simulates $\mathcal{I}$ in the environment $\mathcal{F}_1$ (written $\mathcal{I} \leqslant_{\mathcal{F}_1} \mathcal{M}$) not due to the fact that $\mathcal{F}_1$ is not able to exercise the differing parts of the two systems, but because $\mathcal{F}_1$ cannot distinguish between the outputs (o_1, o_2) produced by $\mathcal{I}$ and $\mathcal{M}$. The $\mathcal{F}_2$ environment distinguishes $\mathcal{I}$ and $\mathcal{M}$, by observing the outputs o_1 and o_2 with two separate transitions.

Relativized simulation is a weaker notion than usual simulation and the perfect vision environment $\mathcal{V}$ is the most discriminating environment:

Proposition 3. *For any two systems $\mathcal{S}_1$, $\mathcal{S}_2$ and for any compatible color-blind environment $\mathcal{E}$ it holds that $\mathcal{S}_1 \leqslant \mathcal{S}_2 \implies \mathcal{S}_1 \leqslant_{\mathcal{E}} \mathcal{S}_2$ and $\mathcal{S}_1 \leqslant \mathcal{S}_2 \iff \mathcal{S}_1 \leqslant_{\mathcal{V}} \mathcal{S}_2$.*

With the above propositions we have hinted at the notion of *discrimination*—the ability of environment to distinguish systems from each other:

Definition 6. A color-blind IOATS $\mathcal{F}$ is more discriminating than $\mathcal{E}$, written $\mathcal{E} \sqsubseteq \mathcal{F}$, iff $\mathcal{F}$ distinguishes more processes: $\mathcal{E} \sqsubseteq \mathcal{F}$ iff $\forall \mathcal{S}_1, \mathcal{S}_2 . \mathcal{S}_1 \leqslant_{\mathcal{F}} \mathcal{S}_2 \Rightarrow \mathcal{S}_1 \leqslant_{\mathcal{E}} \mathcal{S}_2$.

The blind environment $\mathcal{B}$ is the least discriminating—it cannot distinguish any two systems from each other (proposition 2). By proposition 3 the perfect vision environment $\mathcal{V}$ is the most discriminating one.

The notion of discrimination will soon prove fundamental for our developments. We shall use it to design composition operators for behavioral properties, facilitating hierarchical modeling of product lines. Unfortunately the definition of the discrimination is rather abstract. The quantification over all systems, makes it infeasible to reason about it mechanically. To remedy this obstacle we introduce a new preorder on environments: a simulation for color-blind IOATSs.

Definition 7. Let $\mathcal{E} = (\mathit{Out}, \mathit{In}, \mathit{Gen}_{\mathcal{E}}, \mathit{Obs}_{\mathcal{E}}, \xrightarrow{!}_{\mathcal{E}}, \xrightarrow{?}_{\mathcal{E}}, E^0)$ and $\mathcal{F} = (\mathit{Out}, \mathit{In}, \mathit{Gen}_{\mathcal{F}}, \mathit{Obs}_{\mathcal{F}}, \xrightarrow{!}_{\mathcal{F}}, \xrightarrow{?}_{\mathcal{F}}, F^0)$ be color-blind environments. A pair of binary relations, $R_1 \subseteq \mathit{Gen}_{\mathcal{E}} \times \mathit{Gen}_{\mathcal{F}}$ and $R_2 \subseteq \mathit{Obs}_{\mathcal{F}} \times \mathit{Obs}_{\mathcal{E}}$, constitutes a simulation between states of color-blind IOATSs iff $(E, F) \in R_1$ implies that

whenever $E \xrightarrow{i!} e$ then also $F \xrightarrow{i!} f$ and $(f, e) \in R_2$,

and $(f, e) \in R_2$ implies that whenever $f \xrightarrow{O_f?} F$

then also $e \xrightarrow{O_e?} E$ and $O_f \subseteq O_e$ 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 $\mathcal{E}$, written $\mathcal{E} \leqslant \mathcal{F}$, iff $E^0 \leqslant F^0$.

Let $\mathcal{E}$, $\mathcal{F}$ be color-blind environments as in the above definition. Let $\mathbb{S}$ be an endofunction on pairs of binary relations on states of these environments such that:

$$\begin{aligned}\mathbb{S}(R_1, R_2) = (&\\ &\{(E, F) \mid \forall i, e . \exists f . E \xrightarrow{i!} e \implies F \xrightarrow{i!} f \text{ and } (f, e) \in R_2\},\\ &\{(f, e) \mid \forall O_f, F' . \exists O_e, E' . f \xrightarrow{O_f?} F' \implies e \xrightarrow{O_e?} E'\\ &\qquad \text{and } O_f \subseteq O_e \text{ and } (E', F') \in R_1\})\end{aligned}$$

Proposition 4. *A pair of binary relations (R_1, R_2) constitutes a simulation between color-blind IOATSs iff $(R_1, R_2) \subseteq \mathbb{S}(R_1, R_2)$ (pointwise inclusion).*

Proof. $\mathbb{S}$ is a monotonic endofunction on a complete lattice of pairs of binary relations. By Tarski's theorem $\mathbb{S}$ has the greatest fixpoint $\bigcap_{j=0}^{\infty} \mathbb{S}^j(\mathit{Gen}_{\mathcal{E}} \times \mathit{Gen}_{\mathcal{F}}, \mathit{Obs}_{\mathcal{F}} \times \mathit{Obs}_{\mathcal{E}})$, equal to the simulation of Def. 7. Since the fixpoint contains all simulations, it enables use of the known proof technique: to show that an IOATS simulates another, find any simulation relating them. □

The simulation preorder can be established mechanically for finite state systems using state exploration [4]. Thanks to the following central result, these techniques can also be used to verify discrimination properties:

Theorem 1. *For any two color-blind environments $\mathcal{E}$ and $\mathcal{F}$: $\mathcal{E} \sqsubseteq \mathcal{F}$ iff $\mathcal{E} \leqslant \mathcal{F}$.*

We prove the theorem at the state level, which generalizes directly to the IOATS level.

Definition 8. Let $\mathcal{E}$ and $\mathcal{F}$ be color-blind environments with identical signatures and let $E \in \mathit{Gen}_{\mathcal{E}}$, $F \in \mathit{Gen}_{\mathcal{F}}$ be generators. The generator F is more discriminating than E, written $E \sqsubseteq F$, iff for all observers s_1, s_2 of all systems $\mathcal{S}_1$, $\mathcal{S}_2$ (compatible with $\mathcal{E}$, $\mathcal{F}$) $s_1 \leqslant_F s_2$ implies $s_1 \leqslant_E s_2$.

Lemma 1. *For any generators $E \in \mathit{Gen}_{\mathcal{E}}$ and $F \in \mathit{Gen}_{\mathcal{F}}$ it holds that $E \leqslant F \Rightarrow E \sqsubseteq F$.*

Proof. Let $\mathcal{S}_1$, $\mathcal{S}_2$ be systems compatible with $\mathcal{E}$, $\mathcal{F}$. Also let $E \leqslant F$, like in the lemma. We show that for any 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 that $(\leqslant_F) \subseteq (\leqslant_E)$. We proceed in two steps: first we introduce a $\mathit{Gen}_{\mathcal{E}}$-indexed family of relations R such that $(\leqslant_F) \subseteq R_E$, second we argue that R_E is an $\mathcal{E}$-relativized simulation, so $R_E \subseteq (\leqslant_E)$.

$$R_E = \{(s_1, s_2) \in \mathit{Obs}_{\mathcal{S}_1} \times \mathit{Obs}_{\mathcal{S}_2} \mid \exists F' \in \mathit{Gen}_{\mathcal{F}} . E \leqslant F' \wedge s_1 \leqslant_{F'} s_2\}$$

First step: $(\leqslant_F) \subseteq R_E$, since $E \leqslant F$. Second step: 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 $E'' \leqslant F''$.

Lemma 2. *For any generators $E \in Gen_{\mathcal{E}}$ and $F \in Gen_{\mathcal{F}}$ it holds that $E \sqsubseteq F \Rightarrow E \leqslant F$.*

For a set of inputs $I \subseteq In$ and an output $o \in Out$ define a looping system $\mathcal{L}(I, o)$ (Fig. 13). 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 prove the contrapositive: for all $\mathcal{E}$,$\mathcal{F}$ and E, F, their generators, $E \not\leqslant F$ implies $E \not\sqsubseteq F$ (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 $n \geq 1$ such that $(E, F) \in \bigcap_{j=0}^{n-1} \mathbb{S}^j(Gen_{\mathcal{E}} \times Gen_{\mathcal{F}}, Obs_{\mathcal{F}} \times Obs_{\mathcal{E}})$ and $(E, F) \notin \bigcap_{j=0}^{n} \mathbb{S}^j(Gen_{\mathcal{E}} \times Gen_{\mathcal{F}}, Obs_{\mathcal{F}} \times Obs_{\mathcal{E}})$.

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 (2)), 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. 13b. It is easy to observe that $s_1 \leqslant_F s_2$, but $s_1 \not\leqslant_E s_2$.

2°. Inductive step. Now consider that $n > 1$. 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.

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 violation of simulation between E' and F' in $n - 1$ steps. By our induction hypothesis there 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. 13c. It is not hard to see that $s_1 \leqslant_F s_2$, but $s_1 \not\leqslant_E s_2$. □

5 Composition of Behavioral Properties

Typical code generators do not use any context information, assuming that the model is combined with the perfect vision environment $\mathcal{V}$. Another extreme would be a program synthesis tool requiring a precise environment model, imposing a significant burden on engineers. We propose light-weight, composable, partial specifications of environments in the form of behavioral properties like: that certain events always come interleaved (e.g. on/off switch), or that there is causality between an input and an output (e.g. a timer only timeouts after it has been started). Each property can be expressed as a simple color-blind IOATS. In this section we consider ways of composing such properties in a conjunctive and disjunctive manner. Conjunctions express adding up inabilities of environments. If one event is unobservable and another event is unobservable then both are unobservable. Disjunctions express adding up abilities. If one event is observable or another is observable, then both might be observable. This means that conjunction decreases discriminative power, making observation classes coarser in the transition systems representing properties, while disjunction increases discriminative power, making observation classes finer. It turns out that the suitable semantics for both connectives can be constructed using greatest lower bounds (glbs) and least upper bounds (lubs) with respect to the discrimination preorder $\sqsubseteq$.

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 over Out ordered by inclusion forms a complete lattice (and hence the set of all partitionings). 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. Due to the alternating nature of communication in our setup, sums and products alternate too: a sum of generators evolves into a product of observers, and dually a sum of observers evolves into a product of generators.

Definition 9. 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. We define their sum to be a color-blind IOATS $\mathcal{E}+\mathcal{F} = (Out, In, Gen_{\mathcal{EF}}, Obs_{\mathcal{EF}}, \xrightarrow{!}, \xrightarrow{?}, \{E^0, F^0\})$, where $Gen_{\mathcal{EF}} = \mathcal{P}(Gen_{\mathcal{E}} \cup Gen_{\mathcal{F}})$, $Obs_{\mathcal{EF}} = \mathcal{P}(Obs_{\mathcal{E}} \cup Obs_{\mathcal{F}})$ and $\xrightarrow{!}$, $\xrightarrow{?}$ are defined by recursively applying the SG and PO rules given shortly.

Sums intuitively correspond to disjunctions of properties. The composition is synchronous: all composed generators take identical steps simultaneously. From the system's perspective a single input is generated. A sum should be as discriminating as any of the summands. For this reason after taking a generation transition over an input i, the sum advances to a product of generators reachable by i from any of the summands. This product, which also embeds determinization, builds the coarsest

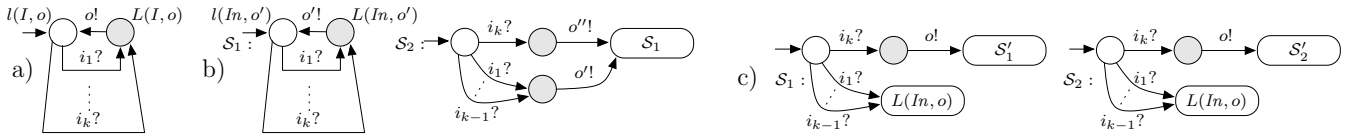

Fig. 13. a) $\mathcal{L}(I,o)$. b) Counter example systems $\mathcal{S}_1$ and $\mathcal{S}_2$. c) $\mathcal{S}_1$ and $\mathcal{S}_2$ created in the inductive step of the proof of lemma 2. Although all these examples assume that $In = \{i_1, \ldots, i_j\}$, the finiteness of In is only a visualization convention and is not relied upon in the proofs.

observation relation that is *finer* than any of the original observation relations, guaranteeing that indeed the constructed observer is as discriminative as necessary:

$$\frac{E_1 \xrightarrow{i!} e_1 \ \ldots \ E_n \xrightarrow{i!} e_n}{\sum_{k=1}^{n} E_k \xrightarrow{i!} \prod_{k=1}^{n} e_k} \text{ SG}$$

$$\frac{O \in \prod_{k=1}^{n} P_{e_k} \quad \mathbb{E} = \{E \mid \exists 1 \leq k \leq n. \exists O' \subseteq Out. e_k \xrightarrow{O'?} E \wedge O \subseteq O'\}}{\prod_{k=1}^{n} e_k \xrightarrow{O?} \sum \mathbb{E}} \text{ PO}$$

Observation classes in the product of observers (PO) are finer than observation classes of any of the composed processes. Whenever any output 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 reachable generators.

The product of two environment IOATSs is defined as follows.

Definition 10. 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. Their product is an IOATS $\mathcal{E} \times \mathcal{F} = (Out, In, Gen_{\mathcal{EF}}, Obs_{\mathcal{EF}}, \xrightarrow{!}, \xrightarrow{?}, \{E^0, F^0\})$, where $Gen_{\mathcal{EF}} = \mathcal{P}(Gen_{\mathcal{E}} \cup Gen_{\mathcal{F}})$, $Obs_{\mathcal{EF}} = \mathcal{P}(Obs_{\mathcal{E}} \cup Obs_{\mathcal{F}})$ and transition relations $\xrightarrow{!}$, $\xrightarrow{?}$ are defined by recursively applying the PG and SO rules given shortly.

A product of generators corresponds to a conjunction of properties (or synchronous composition in CSP [14]). Again the generator rule is very simple and synchronous, but the observer rule this time builds the finest observation relation that is *coarser* than any of the factors:

$$\frac{E_1 \xrightarrow{i!} e_1 \ \ldots \ E_n \xrightarrow{i!} e_n}{\prod_{k=1}^{n} E_k \xrightarrow{i!} \sum_{k=1}^{n} e_k} \text{ PG}$$

$$\frac{O \in \bigsqcup_{k=1}^{n} P_{e_k} \quad \mathbb{E} = \{E \mid \exists 1 \leq k \leq n. \exists O' \subseteq O. e_k \xrightarrow{O'?} E\}}{\sum_{k=1}^{n} e_k \xrightarrow{O?} \prod \mathbb{E}} \text{ SO}$$

Observation classes in the sum of observers (SO) are coarser than classes of any of the composed observers. 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 exceed the number of original observers n.

The result of a composition is a well-formed color-blind IOATS enjoying the following essential property:

Theorem 2. *$\sum_{k=1}^{n}\{\mathcal{E}_k\}$ is the least environment with respect to $\leqslant$, which simulates all summands, while $\prod_{k=1}^{n}\{\mathcal{E}_k\}$ is the greatest environment with respect to $\leqslant$, which is simulated by all the factors.*

Since discrimination and simulation coincide (Thm. 1) $\sqsubseteq$ can replace $\leqslant$ in the above theorem: *The sum of environments is the least discriminating environment, more discriminating than each of the summands. The product is the most discriminating environment, less discriminating than each of the factors.* These in turn are standard expectations about conjunction and disjunction. A conjunction (product) of two properties expressing inability to observe two behaviors, will result in a property expressing inability to observe either. Disjunction (sum) of two properties expressing ability to observe something, results in a property expressing the ability to observe both. See example on Fig. 14.

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:

$$R_1 = \left\{ \left(E_j, \sum_{k \in I} E_k \right) \;\middle|\; \begin{array}{c}\text{for any finite } I \text{ and gen-}\\ \text{erators } \{E_k\}_{k\in I} \text{ and } j \in I\end{array} \right\}$$

$$R_2 = \left\{ \left(\prod_{k \in I} e_k, e_j \right) \;\middle|\; \begin{array}{c}\text{for any finite } I \text{ and obser-}\\ \text{vers } \{e_k\}_{k\in I} \text{ and } j \in I\end{array} \right\} \quad (4)$$

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:

$$R_3 = \left\{ \left(\sum_{k \in I} E_k, F \right) \;\middle|\; \begin{array}{c}\forall \text{ finite } I.\ \forall \{E_k\}_{k\in I}.\\ \forall F.\ \forall k \in I. E_k \leqslant F\end{array} \right\}$$

$$R_4 = \left\{ \left(f, \prod_{k \in I} e_k \right) \;\middle|\; \begin{array}{c}\forall \text{ finite } I.\ \forall \{e_k\}_{k\in I}.\\ \forall f. \forall k \in I.\ f \leqslant e_k\end{array} \right\} \quad (5)$$

The proof of the case for products of generators is dual. □

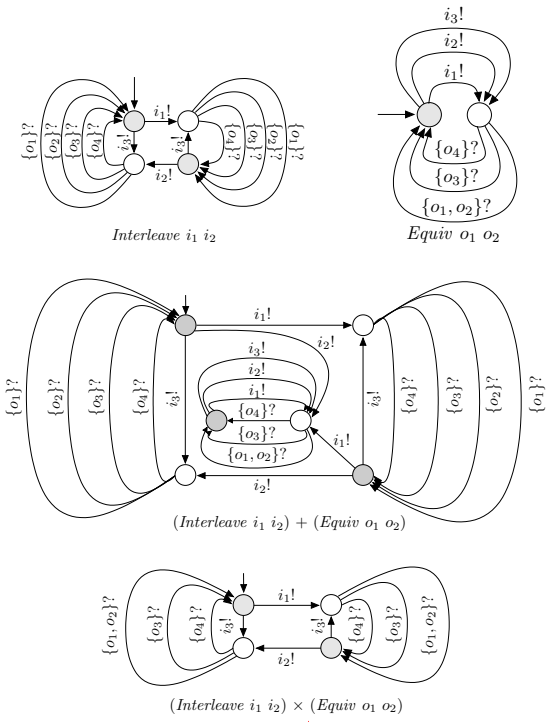

Fig. 14. Environments *Interleave* i_1 i_2 (left) and *Equiv* o_1 o_2 (right), their sum (top) and product (bottom). Transitions to the blind observer **b** are supressed. The product can only generate what both of the factors could generate and distinguish only what both of them could distinguish. The 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.

6 Toward Realistic Design Languages

Until now we have assumed that outputs of systems are atomic. This assumption however often does not hold for realistic languages, which typically support structured output: sets, multisets, sequences or even sequences of sets of atomic actions produced in a single step. We will study two groups of languages. We have successfully applied our framework to the semantics of languages producing sets (state/event systems of section 2, Harel's statecharts [12], synchronous languages [2]) and sequences (Java Card [32], UML state diagrams [29]). Each of these language groups, set based and sequence based, will be discussed in the two following subsections. The set based version is further demonstrated by example in Section 7.

From now on 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 suitable representation for observation classes and computing bounds on classes.

6.1 Set based

State/event models of section 2 are synchronous and non-blocking, so they meet all the assumptions of our framework. The set of abstract events $In = Event$, while the set of abstract outputs contains all possible subsets of *Action*.

$$In = Event \qquad Out = \mathcal{P}(Action)$$

Each state configuration $s \in State$ 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):

$$Obs = State \qquad Gen = State \times State$$

Finally for every global transition $s \xrightarrow{e \quad o} s'$ at the model level we introduce a single observation transition $s \xrightarrow{e?} (s, s')$ and a single generation transition $(s, s') \xrightarrow{o!} s'$ in the abstract IOATS. Remaining states are not related.

These definitions allow us to use the framework of previous sections and model environments for State Event Systems 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 Reduced Ordered Binary Decision Diagrams, or BDDs [3].

We still need to make sure that classes represented on transitions leaving from a single observer are indeed disjoint and form a partitioning (see (2)). 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:

$$\forall i, j \in \{1..n\}.\ i \neq j \ \Rightarrow\ \phi_i \wedge \phi_j \equiv false \qquad (6)$$

$$\phi_1 \vee \ldots \vee \phi_n \equiv true \qquad (7)$$

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. In particular the rules for observers rely on the existence of glbs and lubs for partitionings.

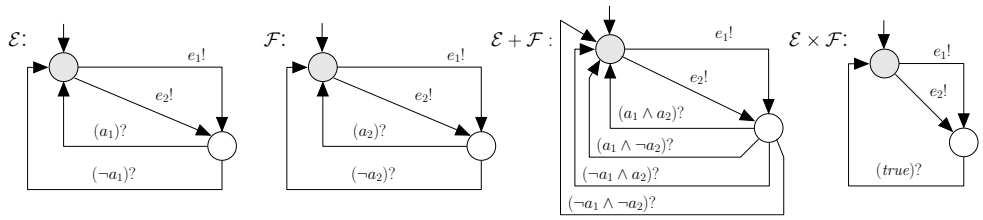

Fig. 15. Left: environments $\mathcal{E}$ and $\mathcal{F}$, suitable for executing set-based reactive systems. Right: the sum and product of $\mathcal{E}$ and $\mathcal{F}$. Observational classes computed according to propositions 5 (for sum) and 6 (for product)

Even though these glbs and lubs exists, we still need to give an efficient way to compute them:

Proposition 5. *Consider two equivalence relations $\sim_\phi$ and $\sim_\psi$ defined on $\mathcal{P}(Action)$, 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 the classic Union-Find algorithm (see [6, 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 6. *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 the union operation.*

Fig. 15 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 a nearly identical adaptation allows applications of our framework to other set-based languages including many hardware description languages, synchronous languages [2] and Harel's statecharts [12].

6.2 Sequence based

Let us now turn from systems producing outputs structured as sets towards systems that produce outputs structured as sequences of atomic actions—for example UML state diagrams. Now each observation transition of the system awaits a single input from the *Event* set, while each generation transition produces an output which is a finite sequence of actions from *Action*:

$$In = Event \text{ and } Out = Action^* \ .$$

The first step in adapting the theory is linking the concrete states of models (for example state configurations in statecharts, or variable store in Java Card) to abstract states of the IOATS. This can normally be done in a direct way (at least for finite state models) in the same spirit as in the previous section. Subsequently the observation and generation relations must be extracted from the semantics of the language in question. Observation classes on the environment side (color-blind) become sets of sequences of actions. Partitioning of $Action^*$ into classes that are regular languages can be described by a finite automaton.

Definition 11. 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 $s \in S$ function $\rightarrow(s)$ is defined for each element of its domain A. We usually write $s \xrightarrow{a} s'$ instead of $\rightarrow(s)(a) = s'$.

A classifier DFA consecutively applies $\longrightarrow$ to a state and the head of the input sequence obtaining a new state and input sequence. An execution over a list of symbols $s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ \ldots \ \xrightarrow{a_n} s_n$ is abbreviated with $s \xrightarrow{a_1 \ldots a_n}{}^* s_n$.

Definition 12. Let $c = (S, A, s, \rightarrow)$ be a classifier. Sequences $\sigma_1, \sigma_2 \in A^*$ are equivalent with respect to c if both advance c to the same state: $\exists s'. s \xrightarrow{\sigma_1}{}^* s' \wedge s \xrightarrow{\sigma_2}{}^* s'$.

The equivalence with respect to a classifier is an equivalence relation and partitions A^* into a finite set of classes, isomorphic with the reachable states.

For a classifier $e = (S_e, Action, s_e, \rightarrow_e)$ consider a mapping of its states to generators $\gamma_e : S_e \rightarrow Gen$. Each

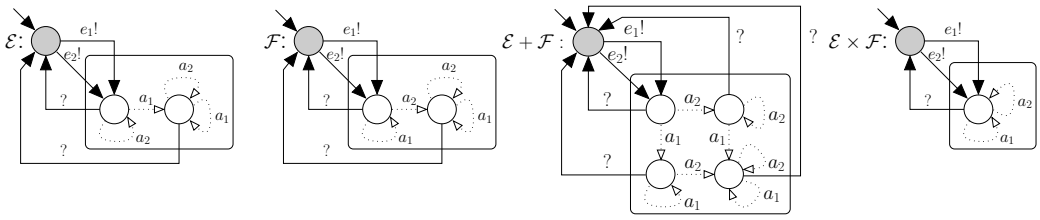

Fig. 16. Environments $\mathcal{E}$ and $\mathcal{F}$ observing sequences, their sum and product.

observer of the environment comprises a classifier and a generator mapping. Environments advance from an observer (e, γ_e) to a generator $\gamma_e(s)$ if it observes a sequence σ advancing the classifier to a state s:

$$(e, \gamma_e) \xrightarrow{\{\sigma \mid s_e \xrightarrow{\sigma}{}^* s\}\ ?} \gamma_e(s) \ .$$

Fig. 16 shows two color-blind IOATSs $\mathcal{E}$ and $\mathcal{F}$ of signature: $Event = \{e_1, e_2\}$ and $Action = \{a_1, a_2\}$. $\mathcal{E}$ distinguishes reactions containing at least one occurrence of a_1 from those 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.

The product of classifiers is a central construction in computing products of observers, supporting composition of environments:

Definition 13. 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 $s_f \xrightarrow{a} s'_f$.

Proposition 7. *Let $\sim_e$ and $\sim_f$ be two equivalences on $Action^*$ induced by classifiers e and f. Their greatest lower bound $\sim_e \sqcap \sim_f$ exists and is induced by $e \otimes f$.*

Fig. 16 presents the sum $\mathcal{E} + \mathcal{F}$ obtained by application of operational rules of section 5 (SG,PO) and the above proposition. $\mathcal{E} + \mathcal{F}$ distinguishes four classes of outputs: an empty sequence, sequences consisting of occurrences of a_1, consisting of occurrences of a_2, and containing occurrences of both a_1 and a_2.

As we have seen before, 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 this 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 $s_1, s_2 \in S$:

$$s_1 \xrightarrow{a} s_2 \text{ if } \exists . p_1 \in S_1 . \exists p_2 \in S_2 .\ p_1 \xrightarrow{a}_e p_2 \text{ or } p_1 \xrightarrow{a}_f p_2$$

The following proposition claims that this function is well-defined, deterministic and input-enabled:

Proposition 8. *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 $s'_2 \in s_2$, 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 9. *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. 16 is a product of $\mathcal{E}$ and $\mathcal{F}$ obtained by application of the composition rules from section 5 (PG,SO) and the above algorithm. This product gives rise to the observer which does not distinguish any sequences.

7 Environment Driven Specialization

We shall now broaden the meaning of a model of a system to encompass a family of systems, and let it represent functionality, which in its entire richness may not be present in any of the actual members being produced. Particular family members will be specified using models of environments, and derived by transformations preserving relativized equivalence in a given color-blind environment. 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 the specialization time. This proof should be obtained automatically using one of the available technologies (type checking, static analysis and model-checking). In this section we firstly formulate correctness condition for transformations and then informally demonstrate a product line derivation scenario, hinting at what techniques could be used to make such automatic derivation viable.

Let T be a model transformation, $\mathcal{M}$ a family model and $\mathcal{E}$ an environment defining a specific family member. Then we will say that T is correct iff the original model and the derived member are in $\mathcal{E}$-relativized two-way simulation relation. $T(\mathcal{M}, \mathcal{E}) \lesssim_{\mathcal{E}} \mathcal{M}$. This means that $\mathcal{E}$ cannot distinguish the behavior of the two systems.

We will now present a family of environment specifications and a corresponding family of alarm clocks derived from the original alarm clock model using the environments. The transition relation of state/event systems (see section 2) produces sets of actions during a single reaction step. In such a setting the observational classes of environments become sets of sets (powersets) of actions.

For a set $A \subseteq Action$ let *ignore* A denote observation classes, which ignore elements of A, but distinguish all the other actions:

$$ignore\ A = \left\{ \{o \cup o' | o' \in \mathcal{P}(A)\} \;\middle|\; o \in \mathcal{P}(Action \setminus A) \right\}$$

Note that ignoring the empty set, *ignore* $\emptyset$, means observing all differences in outputs. Another abbreviation *equiv* A denotes observation classes, which are unable to distinguish between any actions in A:

$$equiv\ A = \left\{ \{o \cup o' | o' \in \mathcal{P}(A) \setminus \emptyset\} \;\middle|\; o \in \mathcal{P}(Action \setminus A) \right\} \cup \mathcal{P}(Action \setminus A)$$

We shall begin with stating general requirements, which hold for all the environments used to execute the alarm clock. These general requirements usually reflect the physical nature of actuators and sensors. In the case of our alarm clock events *dark*/*bright* and *snooze*/*snoozeR* are always generated in an alternating fashion:

$$\mathcal{E}_0 = Interleave\ snooze\ snoozeR \wedge Interleave\ dark\ bright\ .$$

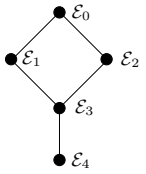

Fig. 17. Relationships between the environments.

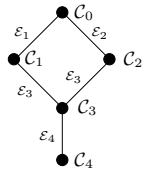

Fig. 18. Relationships between the product variants.

Fig. 8 demonstrates how *Interleave* could be defined using a set-based semantics.

In section 2 the most restricted family member $\mathcal{C}_4$ was introduced in figure Fig. 7. Here we will first introduce some family members that are less restricted.

The least restricted member of the family $\mathcal{C}_1$ is shown in Fig. 4. This model operates in an environment, which becomes blind for the *lightOn* action right after generating the *snooze* event. Formally the environment in which $\mathcal{C}_1$ behaves identical to $\mathcal{C}_0$ is $\mathcal{E}_1 = \mathcal{E}_0 \wedge \mathcal{E}'$, where $\mathcal{E}'$ is defined on Fig. 9.

Fig. 5 presents a new clock $\mathcal{C}_2$, which is devoid of the actual snooze function. The user of this clock can still press the snooze button, but the only effect it has is turning the backlight on for a short while. This user 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. 10.

The third clock variant $\mathcal{C}_3$ is a combination of $\mathcal{C}_1$ and $\mathcal{C}_2$. It has neither the snooze function nor the snooze activated backlight. 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. Note that $\mathcal{C}_3$ 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, 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. 7). The glow-mode lamp is not installed and the *glow* action is reimplemented to turn off the main lamp instead. A corresponding environment $\mathcal{E}'''$ is defined on Fig. 11. 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 characteristics with the restrictions of $\mathcal{E}_3$, giving rise to an even more simple 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. Figures 17–18 present an overview of environments and systems in our product line. Edges represent simulation and relativized simulation. Proposition 10 explains how to interpret transitivity in the hierarchy of systems (Fig. 18).

Proposition 10. *For any 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: $\mathcal{S}_1 \leqslant_{\mathcal{E}} \mathcal{S}_2 \wedge \mathcal{S}_2 \leqslant_{\mathcal{F}} \mathcal{S}_3 \wedge \mathcal{E} \leqslant \mathcal{F} \Rightarrow \mathcal{S}_1 \leqslant_{\mathcal{E}} \mathcal{S}_3$.*

8 Model Transformations

While discussing techniques for implementation of optimizers, we rely as much as possible on existing techniques that are well known in compiler technology, partial evaluation and model-checking. What is important, we believe that optimizations should be designed and implemented for particular languages, not for their abstract semantics. Despite the fact that many transformations can be generalized beyond the particular language, such generalizations rarely lead to successful implementations in specialization tools and compilers. There is very little hope that we can translate any reactive synchronous program into an IOATS, apply the transformations on the abstract layer, and then transform it back to the modeling language, code generate and still obtain significantly optimized code. Contrary—it is most likely that this process of multiple translation will increase the code size significantly. Instead the semantic layer of IOATS should be used for making proofs of correctness of transformations operating on the level of concrete modeling language. We also use the semantic properties, like color-blindness, to inspire our search for transformations.

8.1 Environment Independent Transformations

Many simple optimizations can be achieved just by analysis of the state/event models in absence of any environment, relying on classical data-flow analysis [1, chpt. 17] [27, chpt. 8] and interpreting the model as a control graph. These include constant propagation [27, p. 329], deadcode elimination [27, p. 580], and, to some extent, elimination of side-effect-free code [27, p. 592]. In reactive synchronous models, like state/event systems, these optimizations correspond to simplification of guards, dropping transitions never enabled, dropping states not being targets of any transitions, and dropping entire state machines or processes that produce no side-effects and are never referred to. The notion of side-effect-free part of the model is perhaps a bit less standard than the others, so let us discuss it briefly.

A state machine is *pure* if there are no guards in other machines that refer to it, and it has no outputs on transitions (this is a static property). A set of state machines is *mutually pure* if none of the state machines in the set has any outputs on transitions and none of the state machines outside the set refer to its states in guards. It is not hard to see that a maximal set of mutally pure state machines can safely be removed from the model without *any* execution environment ever observing this change. Formally this can be argued by making a two-way relativized simulation proof in our framework, assuming a perfect vision environment. Note that we have removed the *sensor* component from the alarm clock model using precisely this reasoning (see $\mathcal{C}_4$, Fig. 7).

Typically, all optimizations mentioned above are not applicable for reactive models when used in isolation: programmers rarely write code that is dead, does not compute anything or makes a complex computation that always results in the same constant. In most of such cases these are manifestations of errors, which should be reported. Basic optimizations are nevertheless essential, when combined with more advanced techniques described below, which often produce code with constants in expressions, unreachable states and transitions, etc.

8.2 Optimizations with Static Environments

Knowledge of static properties of environment may allow to apply the above optimizations more aggressively. If some events can never be input to the system, then they can obviously invalidate some guards (a never constraint on an event can be propagated as false to guard conditions and constant propagation may be applied subsequently). If some outputs are never being observed (always ignored), they can be erased from transitions' actions, which may introduce mutually pure components in the model, that could be ereased subsequently as described above.

Perhaps the most interesting optimizations, which are also new, can be performed for static environments exhibiting color-blindness (*Equiv* constraints). If some actions are always equivalent, we can definitely substitute one for another. Remember that actions correspond to invocations of actuator drivers, so if we could eliminate one driver/handler it can potentially save a significant amount of program memory. In order to justify the choice, one needs more information about the properties of drivers. In the following we assume that each action $a_i \in \mathit{Action}$ has a weight w_i assigned describing a measure to be minimized (execution time, code size, price of the respective actuator, etc). We formulate the problem as follows:

Input: a set of actions $\mathit{Action} = \{a_1, \ldots, a_n\}$, corresponding weights $w_1, \ldots, w_n$, and an equivalence relation E on Action.

Output: An optimal selection of equivalence class representatives, so that when substituted in the model for all call places they minimize the total cost for the model:

$$\sum_{i=1}^{n} x_i w_i \ ,$$

where $x_i = 1$ if a_i has been chosen as a representative for some class, and $x_i = 0$ otherwise. This problem can be solved computing equivalence classes using a UNION-FIND like algorithm, and then choosing the

cheapest element for each class. Again proofs of correctness of the above optimizations can be written using two-way relativized simulation as a requirement, assuming an IOATS environment that represents the static properties required (all such environments have exactly one generator state and one observer state).

8.3 Optimizations with Dynamic Environments

There is no doubt, that introducing behavior in environments, vastly increases the amount of possible optimizations. For example precise reachability analysis can be executed on the model, to achieve more aggressive kind of code elimination, or identify more parts of guards as being constants. Nevertheless we shall not discuss all these possibilities here. Instead, we focus on a transformation that directly exploits color-blindness of our environments. Ultimately we would like to consider an algorithm for Action-Set-Minimization, an extension of the one presented in the previous paragraph, but with the equivalence relation on actions changing dynamically. This complicates the problem significantly, as now each transition, not each action, may require choosing a different substitute for its actions. Moreover this substitute candidate has to satisfy observation requirements of all environment transitions that can possibly be executed in parallel with the given system transition.

We can use reachability analysis [24,4] to find pairs of environment/system transitions that can fire simultaneously. We are particularly interested in identifying pairs containing one observation transition of environment and one generation transition of the system. During the model exploration one should record information about what sets of actions are observed in what observation classes of environments, ultimately gathering a list of pairs: each consisting a subset of call places and a formula describing the observation class. This information can be translated into a constraint satisfaction problem, with the objective to minimize a cost function (most typically reflecting the code size as above). An off-the-shelf tool like ILOG's CPLEX [18] could be used to solve the problem, returning for each set of actions a cheaper replacement for this set. In fact, as it is typical in industrial optimization problems, the optimality of the solution is not required for the correctness of our application, so an approximating solver could be used (for example LP (Linear Programming) based), or the ILP (Integer Linear Programming) solver can be interrupted early to use the best solution found so far, if finding the optimal solution proves intractable. Finally the information obtained from the solver is used to restructure the transition layout and actual calls placed on transitions. Ultimately less drivers are needed in the final product, and the kernel code is smaller.

9 Related Work

Derivation of product lines is often associated with partial evaluation [19,8,13]. There have been approaches to enable partial evaluation based on execution traces instead of fixed input values [15,28,10], nevertheless they were never implemented for realistic languages. We fear that these transformations, designed for abstract process calculi, can be barely applied in such contexts. Our framework allows more transformations than known before due to the color-blindness, which allows some non-reductive mutations in the program.

Wąsowski [36] presented a static framework for specifying environments for reactive models, which relies solely on state independent properties. The present paper, itself an extension of [23], provides a theoretical foundation for a product line management setup similar to Wąsowski's [36], but based on behavioral properties.

Relativized simulation has been introduced by Larsen in [22,21,20]. Our framework is modeled after this work, rephrased in the setting of IOATSs and extended with color-blindness. In Larsen's formulation, based on simple labeled transition systems [26], it was impossible to directly express an environments' inability to distinguish outputs. Further results on color-blind environments, as well as detailed proofs, can be found in [37].

The study of systems embedded into behavioral contexts is quite mature [21,9,25,30,17]. Our work stems out from the field, by its direct 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.

Czarnecki and Antkiewicz [7] present a dual approach to product line derivation. We specify variants by legal use (black-box), they specify variants by annotating model internals with conditions (white-box). In our case the derivation is difficult, but safety properties are preserved. In their case the derivation is relatively easier, while specifications still need to be verified.

10 Conclusion & Future Work

We have presented the semantics of a specification language for environments of reactive synchronous systems, together with a notion of context-dependent refinement based on color-blindness. This refinement relation is more liberal than usual in allowing some mutations to program outputs, instead of bare reductions. We have explained and demonstrated how partial specifications of behaviors can be composed and used to define families of products. The framework was designed as a core of an upcoming tool for compact code generation and product line derivation for discrete control embedded systems. Our specifications shall be used as preconditions for advanced model optimizers/specializers. We have thoroughly discussed issues, which arise in the implementation of the

theory for realistic languages, especially focusing on languages with sequences as outputs.

An implementation [31] of a powerful context-aware optimizer for models based on model-checking and program analysis is planned. This prototype tool is supposed to be compatible with an industrial development environment for embedded systems [16], which will allow for realistic case studies.

References

1. Andrew A. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.
2. Gérard Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction. Essays in Honour of Robin Milner*, pages 425–454. MIT Press, Cambridge, MA, 2000.
3. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986.
4. Edmund M. Clarke. *Model Checking*. MIT Press, 1999.
5. Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.
6. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.
7. Krzysztof Czarnecki and Michał Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In R. Glück and M. Lowry, editors, *Generative Programming and Component Engineering (GPCE)*, volume 3676 of *LNCS*, pages 422–437. Springer-Verlag, 2005.
8. Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Partial Evaluation*, volume 1110 of *LNCS*, Dagstuhl Castle, Germany, February 1996. Springer-Verlag.
9. Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Symposium on Foundations of Software Engineering (FSE)*, pages 109–120. ACM Press, 2001.
10. Sandro Etalle and Maurizo Gabbrieli. Partial evaluation of concurrent constraint languages. *ACM Computing Surveys*, 30(3es), September 1998.
11. Hassan Gomaa. *Design Software Product Lines with UML*. Addison-Wesley, 2001.
12. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
13. John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors. *Partial Evaluation: Practice and Theory. International Summer School*, volume 1706 of *LNCS*. Springer-Verlag, 1999.
14. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
15. Haruo Hosoya, Naoki Kobayashi, and Akinori Yonezawa. Partial evaluation scheme for concurrent languages and its correctness. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96*, volume 1123 of *LNCS*, pages 625–632. Springer-Verlag, 1996.
16. IAR visualSTATE®. `www.iar.com/Products/VS`.
17. A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. In *POPL 2001*. ACM Press.
18. Ilog CPLEX. `www.ilog.com/products/cplex`.
19. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
20. K.G. Larsen and R. Milner. A compositional protocol verification using relativized bisimulation. *Information and Computation*, 99(1):80–108, 1992.
21. Kim G. Larsen. *Context Dependent Bisumulation Between Processes*. PhD thesis, Edinburgh University, 1986.
22. Kim G. Larsen. A context dependent equivalence between processes. *Theoretical Computer Science*, 49:184–215, 1987.
23. Kim G. Larsen, Ulrik Larsen, and Andrzej Wąsowski. Color-blind specifications for transformations of reactive synchronous programs. In M. Cerioli, editor, *Fundamental Approaches to Software Engineering (FASE)*, volume 3442 of *LNCS*. Springer-Verlag, 2005.
24. J. Lind-Nielsen, H. R. Andersen, H. Hulgaard, G. Behrmann, K. Kristoffersen, and K. G. Larsen. Verification of large state/event systems using compositionality and dependency analysis. *Formal Methods in System Design*, 18(1):5–23, 2001.
25. Nancy Lynch. I/O automata: A model for discrete event systems. In *Annual Conference on Information Sciences and Systems*, pages 29–38, Princeton University, 1988.
26. Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.
27. Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
28. Masaki Murakami. Partial evaluation of reactive communciating processes using temporal logic formulas. In *Workshop on Algebraic and Object-Oriented Approaches to Software Science*, 1995.
29. Object Management Group. OMG Unified Modelling Language specification, 1999. `www.omg.org`.
30. Sriram K. Rajamani and Jakob Rehof. Conformance checking for models of asynchronous message passing software. In E. Brinksma and K. G. Larsen, editors, *Conference on Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 166–179. Springer-Verlag, 2002.
31. Scope. `www.itu.dk/~wasowski/scope`.
32. Sun Microsystems. Java card(TM) specification. `java.sun.com/products/javacard/specs.html`.
33. Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
34. Jan Tretmans. Test generation with inputs, outputs, and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.
35. 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 *LNCS*, pages 278–297. Springer-Verlag, 1990.
36. Andrzej Wąsowski. Automatic generation of program families by model restrictions. In *Software Product Line Conference (SPLC)*, volume 3154 of *LNCS*. Springer-Verlag, 2004.
37. Andrzej Wąsowski. *Code Generation and Model Driven Development for Constrained Embedded Software*. PhD thesis, IT University of Copenhagen, January 2005.