Modal I/O Automata for Interface and Product Line Theories

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

[1] Department of Computer Science, Aalborg University
[2] Computational Logic and Algorithms Group, IT University of Copenhagen
{kgl,ulrik,wasowski}@cs.aau.dk

Abstract. Alfaro and Henzinger use alternating simulation in a two player game as a refinement for interface automata [1]. We show that interface automata correspond to a subset of modal transition systems of Larsen and Thomsen [2], on which alternating simulation coincides with modal refinement. As a consequence a more expressive interface theory may be built, by a simple generalization from interface automata to modal automata. We define modal I/O automata, an extension of interface automata with modality. Our interface theory that follows can express liveness properties, disallowing trivial implementations of interfaces, a problem that exists for theories build around simulation preorders. In order to further exemplify the usefulness of modal I/O automata, we construct a behavioral variability theory for product line development.

1 Introduction

An interface theory [1,3,4,5,6,7] is a type-system-like theory for component languages, where types (*interfaces*) describe components (*implementations*) with *composition* being the only operator available. A type error proves that either a component does not *conform* to its interface, or that two composed components are *incompatible*. Since the overall structure of these type systems is so simple, it is often accepted not to give typing rules explicitly when describing interface theories (for example [1,3,4,5,6]), focusing instead on the essential ingredients of conformance, compatibility and composition.

Regular, non-component types are only applied to existing objects in program code. In contrast for interface theories it makes sense to discuss interfaces as specifications of application's architecture in isolation from actual source code. An interface abstracts the component in terms of the assumptions made by the component and the guarantees that it provides. One reasons about possible connections between component implementations (*compositions*) by using properties of composition of interfaces; most importantly *independent implementability* (that any implementations conforming to compatible interfaces are compatible) and generality properties (that the composition of interfaces produces an interface with the weakest assumptions and strongest guarantees).

R. De Nicola (Ed.): ESOP 2007, LNCS 4421, pp. 64–79, 2007.

We consider behavioral interface theories suitable for specification of communication protocols between components (web services or embedded systems). Such theories typically require a *contravariant* treatment of inputs and outputs to ensure deadlock-free implementations: inputs guaranteed by the specification are always offered by the implementation and that the implementation never produces more outputs than the specification. This observation led de Alfaro, Henzinger and colleagues [1,3,4] to a conclusion that game theoretical models of interaction are most suitable as building blocks for behavioral interface theories. While we do appreciate the values of the game theoretical formulations, we disagree with some claims in the above cited work and argue that game formulations are insufficient in themselves: there is a genuine value in combining the game theoretical approach with more traditional formulations based on transition systems, or more precisely on modal transition systems.

The two worlds of game models and modal transition systems convey largely orthogonal information about the moves of a system. Game models specify who has *control* over transitions, while modal transition systems focus on requirements, *modality*: which moves are allowed and which are required. In this paper we try to relate the two worlds, explain their weaknesses and their qualities. Eventually we combine them into a unified interface theory.

Game theoretical notions of conformance are often based on alternating simulation [8]. We show that alternating simulation in a two player setting, as used in interface automata [1,9], is just a special case of modal transition systems refinement developed by Larsen and Thomsen [2] in the late eighties. This suggests that the real value of the game theoretic approach to component theories does not lie in the use of alternating simulation, but in the use of *control* information in the composition synthesis algorithms.

Not surprisingly then, modal transition systems themselves cannot be used to build an interface theory, without adding control information. We build a new interface theory around *modal I/O automata,* which combine features of both game theoretic models and modal transition systems. Thanks to this new combination, our interfaces are now able to express liveness properties, which was impossible in existing interface theories (after this work has been completed we have learned about [10], which achieves a similar effect in a different setting).

In order to further demonstrate the usefulness of our modal I/O automata, we construct a *product line* [11,12,13] *theory.* In simple words a product line is a set of similar products built by combining *assets* from a common platform available in the development process. The differences between the products are referred to as *variability.* Our theory is a behavioral formalism for describing the variability of components. The theory supports deciding whether given requirements can be satisfied by choosing concrete instances from the set of available assets. This theory, though very small, is to the best of our knowledge one of the very few attempts at describing software product lines in a behavioral fashion, and unlike the previous work [14], which takes a top-down approach to describing product families, it facilitates a bottom up construction of products, which is how product line development is more typically understood in the software engineering

community. This contribution is not meant to be comprehensive, highly developed and well set in the tradition of the product line development. It should be understood as a simple example that emphasizes the semantic difference between modeling components in component based development and modeling assets for product family development. We do hope to extend this theory soon and report about it separately in detail.

The paper proceeds as follows. In the next section we shall explain the main results of the paper in nontechnical terms. Our main results concentrate in sections 3, 5 and 6. In Section 3 we draw a correspondence between the alternating simulation and observational modal refinement. In Section 4 modal I/O automata are defined, which are then used to construct an interface theory in Section 5 and a product line theory in Section 6. Sections 5 and 6 are largely independent, though they share a lot of intuitions. We conclude in Section 7.

2 Interface Automata vs Modal Automata: An Example

Consider an example interface automaton for a *Client* component (Fig. 1 (left), originally presented in [1]). This simple model describes a component that occasionally may want to send a package, and once it has made the request it is ready to receive an acknowledgment. The signature of the interface also mentions a fail input, but the component is never able to receive it. This means that *Client* is only capable of interacting with network links that never fail.

In interface automata, due to a game theoretic semantics, all outputs are controlled by the component itself (called the *Output* player), while all inputs to such components are controlled by the environment player (called the *Input* player). An implementation conforms to the interface iff whenever some input is offered by the interface, then it is also offered by the implementation, and whenever an implementation produces any output, this output is also present in the interface (conformance formalized as alternating simulation [8]).

Such a notion of conformance implies that compatibility can be passed from interfaces to components: if there is no winning strategy for the input player that leads to a deadlock in the interface automaton, then there won't be such a strategy for the same player that interacts directly with any implementation. Similarly if there is no strategy for the output player that leads to an output that cannot be accepted by the environment, then there is also no such strategy for any of the implementations.

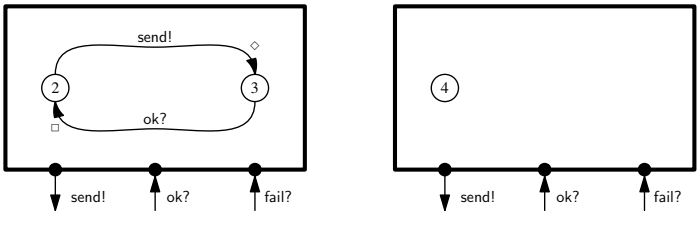

Fig. 1. The *Client* interface (left) and a trivial implementation of it (right)

Unfortunately this notion of conformance, though very much safety oriented, does not enforce that the implementations take on any useful activities at all. Consider for example the diagram on the right side of Fig. 1. It presents a model of an implementation that does not perform any actions ever. In other words this is a network application that does not use the network at all. Still this new model conforms to its interface on the left, as in its initial state it does not add any illegal outputs and it offers all the inputs that were offered by the interface.

If we turn this into the terminology used in modal transition systems it means that all the inputs are *required*, which is indicated by the $\Box$ (must) modality on the corresponding transition, and the outputs are *allowed*, which is indicated by the $\Diamond$ (may) modality on the transitions. In a modal transition systems perspective, conformance is based on modal refinement [2]. This refinement requires that whenever an implementation makes a step, then it must be possible to mimic it by an allowed transition of the specification; whenever the specification makes a required step it must be possible to match it with some required step of the corresponding state in the implementation. With the assignment of *may* to output transitions and *must* to input transitions this sounds nearly like the alternating simulation described above. In Section 3 we prove that indeed the two relations coincide if we require that the may transition relation is input-enabled.

Consequently modality gives strictly more modeling power than alternating refinement. Various modalities can be assigned to actions regardless of whom controls them. Instead of allowing all possible extensions on inputs, as in interface automata, the designer is able to control what extensions are allowed. For example we can change the *Client* model of Fig. 1 to have a must modality ($\Box$) on the *send!* transition, which will have the effect that now all the implementations must be able to proceed producing an output. This would rule out trivial implementations as the one presented on the right side of Fig. 1.

The game theoretic formulation of conformance gives a certain interpretation to inputs and outputs. Namely that inputs are *incoming requests* for service (for example remote procedure calls), while outputs are *outgoing requests* for service (also remote procedure calls, albeit in the other direction). With such an interpretation it becomes clear that removing services from the promised list should be illegal, while removing calls to external services is perfectly fine. This is exactly what alternating simulation achieves. What it misses is a more complex structure of communication.

In asynchronous systems some messages indeed convey calls for service, however many other return feedback from the services (return a value). When a given output models returning a value from a component, then clearly it should never be removed, as then the whole component becomes useless. Fig. 2 illustrates another interface modeling a data link layer, which exploits the interplay between control and modality. The *must* modality is placed on transmt! transitions, as the data link layer would be useless if the implementation was permitted not to forward packets down the stack. Similarly the transition sending back the error message cannot legally be removed. At the same time the call for linkStatus! is a may transition as some implementations are allowed not to consult the hardware

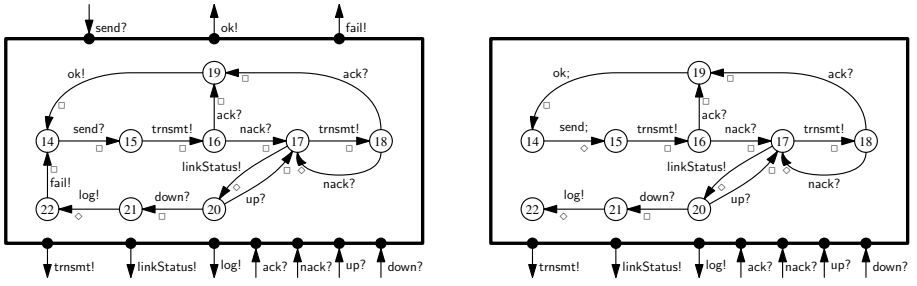

Fig. 2. *DataLink* layer with nontrivial modalities (left). Composition *DataLink* $\otimes$ *Client* (right). State 22 is an error state, where *DataLink* can produce the fail action, not accepted by *Client*.

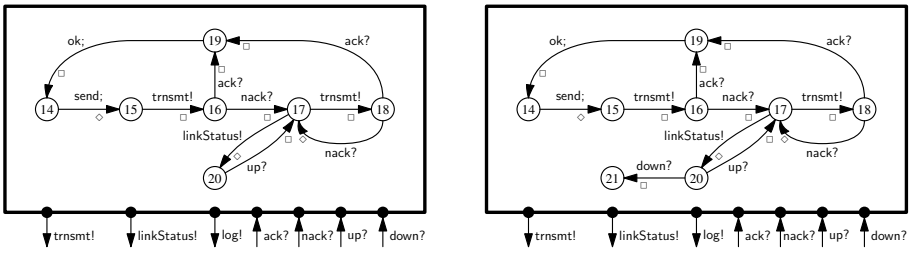

Fig. 3. Composed interfaces *LinkLayer* | *Client* and variability models *LinkLayer* · *Client*

link explicitly to detect errors. Finally not all implementations are forced to be able to work with links that fail twice in a row, which is modeled by the second nack! transition being a may transition.

Now consider how the two interfaces of Fig. 1 (left) and Fig. 2 (left) should be composed. The composition resembles a product computation (taken separately for the may transition relation and the must transition relation). As a result we obtain the interface presented on the right side of Fig. 2. Because the client component was so weak, the ultimate interface shows a system that possibly may never do anything. However if *Client* will send some packets, these packets will certainly be processed by the composition, unless the hardware link is broken. In such a case it might be that the implementation will produce a fail! message which will cause a deadlock with the current version of the *Client* (this can happen when the composition is in state 22). Since we cannot modify the composed system we instead synthesize a new interface which restricts the use of the composition in order to guarantee error freeness. States of the composition that can experience deadlocks are called *error states*. We follow Alfaro and Henzinger in removing error states, and transitively all states from which error states can be reached by following *internally controllable transitions* of the component (outputs and internal actions). This leads to the interface on Fig. 3

(left), expressing the fact that this component works well as long as the physical link never goes down.

The pruning mechanism described above would not be possible without the information describing which transitions are internally controllable being explicitly present in the model. It does not seem possible to compute the safe fragment of the product automaton, by just investigating the modalities of transitions. While we have said that modal refinement is strictly more expressive than alternating simulation, the control information of interface automata has its unique qualities too: it enables valuable synthesis algorithms not otherwise possible.

Let us now revisit the model of Fig. 2 (left) giving it a different interpretation than previously. Instead of perceiving it as an abstraction of a component, we should now see it as a description of a set of components. A modal automaton describes in fact a whole, often infinite, set of possible implementation automata[1]. One can think of them as all possible configurations of the model. This feature of modal automata suggests the possibility of using them as a behavioral formalism in describing variability in product lines.

A product line is a collection of products that are similar in that they offer overlapping functionality, and in that they are built from assets selected from a common platform. In here we want to describe both assets and the whole product line by modal I/O automata. If each of the assets is modeled as a modal I/O automaton we can model the capabilities of the family by composing these descriptions. However this time we would not be interested in a composition that guarantees compatible behavior of any selection of assets. It is normally expected that not all the assets in a product line platform are mutually compatible. Some of them will deadlock (for example a failing link layer and our *Client* component). The requirement for composing the variability descriptions is not to synthesize an interface that guarantees correctness of composition of all possible combination of assets, but to precisely describes what the correct combinations are: i.e. what are the deadlock free behaviors respecting the modalities that can be constructed with the available automata.

It turns out that a composition like that exists and it resembles the pruning of the product automaton for interface automata. The only difference is that now error states are the states where the error must be possible to realize (so one party must be required to produce an output that the other party must not be allowed to receive) and that we prune all the states from which reaching an error state is unavoidable (in our interface theory we have pruned states from which reaching errors might be possible).

The result of composing *Client* and *LinkLayer* using the variability model semantics is presented on the right side of Figure 3. This result contains a slightly bigger model than the interface automaton composition on the left. It states that there exists a pair of assets (implementations of *Client* and *LinkLayer*) such that it is able to accept a link down message without an error message. The transition

[1] This is also true for interface automata, though to a much lesser extent. Due to the lack of modality the set of implementations for an interface automaton is much simpler than it can be for a modal automaton.

with the down message was removed in the interface compositions as, for some pairs of implementations, it would lead to a deadlock.

Can a given specification be implemented by choosing components from available assets? Is the result of the composition the most general possible, containing all possible legal products? Can we find what the configuration of these elements should be? We address some of these questions in section 6, with an intention of elaborating more in upcoming work.

3 Alternating Simulation vs Modal Refinement

Let us begin with defining modal automata, a version of modal transition systems [2] extended with signatures. A modal automaton has two transition relations indicating respectively allowed (*may*) and required (*must*) behavior.

Definition 1 (Modal Automaton). *A modal automaton S is a six tuple: $S = (states_S, start_S, ext_S, int_S, \longrightarrow_\Diamond, \longrightarrow_\Box)$ where $states_S$ is a finite set of states, $start_S \in states_S$ is the initial state, ext_S and int_S are disjoint sets of external and internal actions and $act_S = ext_S \cup int_S$, $\longrightarrow_{\Diamond S} \subseteq states_S \times act_S \times states_S$ is the may transition relation describing allowed behavior, and $\longrightarrow_{\Box S} \subseteq states_S \times act_S \times states_S$ is the must transition relation describing required behavior.*

Throughout the paper we sometimes use the symbols "!", "?" and ";" after an action. This is done in order to increase the readers intuition of whether the action is respectively an output, input or internal action. No symbol is used when the action can be of more than one type. These symbols could be left out completely as it is the identity of the action that is significant.

In the following we write $s \xrightarrow{\tau}{}^*_\Box s'$ meaning that there exists a sequence of internal *must* actions leading from s to s'. The same is defined for *may* transitions.

A modal automaton is *syntactically consistent* if everything that is required is also allowed, such that $\longrightarrow_\Box \subseteq \longrightarrow_\Diamond$. In the following we only consider syntactically consistent modal automata. A modal automaton is an *implementation* if the two transition relations coincide.

A modal automaton describes a set of possible implementations. Simplistically when refining a modal automaton specification into an implementation one can remove a *may* transition, that does not have a corresponding *must* transitions or strengthen it into a *must* transition. In general this refinement is not syntactic, but behavioral, so it is not the syntactic transitions that are refined but the actual steps taken by the transition system. The same transition can be refined differently each time it is taken.

Definition 2 (Modal Refinement). *For a pair of modal automata S and T with the same signature, a binary relation $R \subseteq states_S \times states_T$ is a modal refinement if whenever sRt and $a \in act_S$ it holds that*

if $t \xrightarrow{a}_\Box t'$ then $\exists s'.s \xrightarrow{a}_\Box s'$ and $(s', t') \in R$.
if $s \xrightarrow{a}_\Diamond s'$ then $\exists t'.t \xrightarrow{a}_\Diamond t'$ and $(s', t') \in R$.

Modal refinement $\leq_{\mathrm{m}}$ *is defined as the largest such relation. We say that a modal automaton* S *modally refines a modal automaton* T*, written* $S \leq_{\mathrm{m}} T$*, iff there exists a modal refinement containing* $(start_S, start_T)$*.*

Observational modal refinement is a weaker refinement in which the two modal automata can take internal transitions, that cannot be directly observed by the other automaton. In absence of internal actions the observational refinement coincides with the non-observational one.

Definition 3 (Observational Modal Refinement). *For a pair of modal automata* S *and* T *with the same signature, a binary relation* $R \subseteq states_S \times states_T$ *is an observational modal refinement if whenever* sRt *and* $a \in act_S$ *it holds that*

if $t \xrightarrow{a}_{\Box} t'$ *and* $a \in ext_T$ *then* $\exists s'.\, s \xrightarrow{a}_{\Box} s' \wedge (s', t') \in R$.
if $s \xrightarrow{a}_{\Diamond} s'$ *and* $a \in ext_S$ *then* $\exists t'.t \xrightarrow{\tau}{}^*_{\Diamond} t'.\exists t''.t' \xrightarrow{a}_{\Diamond} t'' \wedge (s', t'') \in R$.
if $s \xrightarrow{a}_{\Diamond} s'$ *and* $a \in int_S$ *then* $\exists t'.t \xrightarrow{\tau}{}^*_{\Diamond} t'.(s', t') \in R$

Observational modal refinement $\leq^*_{\mathrm{m}}$ *is defined as the largest such relation. We say that a modal automaton* S *observationally refines a modal automaton* T *if there exists an observational modal refinement containing* $(start_S, start_T)$*.*

Interface Automata [1] can be considered a subset of modal automata in which the external actions ext_S are partitioned into inputs in_S and outputs out_S.

Definition 4 (Interface Automaton). *An* interface automaton P *is a tuple* $P = (states_P, start_P, in_P, int_P, out_P, \rightarrow_P)$ *where* $states_P$ *is a finite set of states,* $start_P \in states_P$ *is the initial state,* in_P*,* out_P *and* int_P *are three pairwise disjoint sets of input, output and hidden (internal) actions respectively, and* $\rightarrow_P \subseteq states_P \times act_P \times states_P$ *is the set of transitions where* $act_P = in_P \cup out_P \cup int_P$*.*

We require that the transition relation is input-deterministic such that for all $s, s', s'' \in states_P$ *and all input actions* $a \in in_P$ *if* $s \xrightarrow{a?} s'$ *and* $s \xrightarrow{a?} s''$ *then* $s' = s''$*.*

Similarly as for Modal Automata we define $s \xrightarrow{\tau}{}^* s'$ for Interface Automata to mean that there exists a sequence of internal transitions leading from s to s'. We define *alternating simulation* for interface automata as commonly used in software specification [9], which is slightly less general than the original [1]:

Definition 5 (Alternating Simulation). *For a pair of interface automata* S *and* T *with the same signature, a binary relation* $R \subseteq states_S \times states_T$ *is an alternating simulation if whenever* sRt *and* $a \in act_S$ *it holds that:*

if $t \xrightarrow{a?} t'$ *and* $a \in in_T$ *then* $\exists s'.s \xrightarrow{a?} s'$ *and* $(s', t') \in R$
if $s \xrightarrow{a!} s'$ *and* $a \in out_S$ *then* $\exists t'.t \xrightarrow{\tau}{}^* t'.\exists t''.t' \xrightarrow{a} t''$ *and* $(s, t'') \in R$
if $s \xrightarrow{a;} s'$ *and* $a \in int_S$ *then* $\exists t'.t \xrightarrow{\tau}{}^* t'$ *and* $(s', t') \in R$

Alternating simulation $\leq_{\mathrm{a}}$ *is defined as the largest such relation. We say that* S *simulates* T*, written* $S \leq_{\mathrm{a}} T$*, if there exists an alternating simulation containing* $(start_S, start_T)$*.*

In order to compare interface automata with modal automata, we construct a translation function $\mathcal{T}$ mapping from the former to the latter. The result of the translation always fulfills the conditions listed below. It is easy to see that for modal automata that fulfill these conditions a reversed mapping can be constructed, too.

1. The may transition relation is input enabled, meaning that for each state $s \in states_S$ and each input action $a \in in_S$ there exists a state s' and a may transition $s \xrightarrow{a?}_{\diamond} s'$
2. The constructed modal automaton is syntactically consistent: $\longrightarrow_{\Box} \subseteq \longrightarrow_{\diamond}$
3. Must transitions are only labeled by inputs: $\longrightarrow_{\Box S} \subseteq states_S \times in_S \times states_S$

Let s_{mayall} be a fresh state that allows all behavior but does not require any behavior. If U denotes the universe of all inputs, such that for all interface automata P, $in_P \in U$, then we define the translation function as follows:

$$\mathcal{T}(states_P, start_P, in_P, out_P, int_P, \longrightarrow_P) = (states_S, start_S, ext_S, int_S, \longrightarrow_{\diamond}, \longrightarrow_{\Box})$$

where $states_S = states_P \cup \{s_{mayall}\}$, $start_S = start_P$, $ext_S = U \cup out_P$, $int_S = int_P$
and $s_1 \xrightarrow{a}{}^{S}_{\diamond} s_2$ if $s_1 \xrightarrow{a}{}^{P} s_2$ and $a \in out_P \cup int_P$
and $s_3 \xrightarrow{a}{}^{S}_{\Box} s_4$ and $s_3 \xrightarrow{a}{}^{S}_{\diamond} s_4$ if $s_3 \xrightarrow{a}{}^{P} s_4$ and $a \in in_P$
and $s_3 \xrightarrow{a}{}^{S}_{\diamond} s_{mayall}$ if $\forall s' \in states_P(s_3, a, s') \notin \longrightarrow^P$ and $a \in U$,
and s_{mayall} is a fresh state such that $\forall a \in act_S . s_{mayall} \xrightarrow{a}{}^{S}_{\diamond} s_{mayall}$.

Theorem 6. *Alternating simulation and observational modal refinement coincide for interface automata in the following sense:*

$$\text{for any two interface automata } S, T\text{: } S \leq_{\mathrm{a}} T \text{ iff } \mathcal{T}(S) \leq^{*}_{\mathrm{m}} \mathcal{T}(T) \qquad (1)$$

Theorem 6 suggests that the usefulness of game theoretical models for component theories does not lie in its conformance relation. The crux is the use of control information in synthesis algorithms, when paths to error states are pruned. If this is the case we can construct an interface theory based on modal refinement and modal automata augmented with control information. Since modal refinement is richer and we can use a generalization of the synthesis algorithm used for interface automata, we will obtain a more expressive interface theory.

The fact that alternating simulation coincides with the *observational* version of modal refinement is expected, because Definition 5 embeds a closure on internal transitions. In fact in the absence of internal actions alternating simulation coincides with the regular modal refinement, as described in Definition 2, which is easy to prove. In order to simplify the developments we use the regular modal refinement ($\leq_{\mathrm{m}}$) from now on, even though most of our theorems can reasonably be considered for the observational refinement ($\leq^{*}_{\mathrm{m}}$), too.

4 Modal I/O Automata

Let us now define modal I/O automata, an extension of modal automata with control information, that will be the main ingredients of our interface theory and the product line theory coming in the next sections.

Definition 7. *A modal I/O automaton S is a tuple $S = (states_S, start_S, in_S, out_S, int_S, \longrightarrow_\Diamond, \longrightarrow_\Box)$, where $states_S$ is a set of states, $start_S \in states_S$ is an initial state, in_S, out_S and int_S are pairwise disjoint sets of inputs, outputs and internal actions respectively ($act_S = in_S \cup out_S \cup int_S$), $\longrightarrow_{\Diamond S} \subseteq states_S \times act_S \times states_S$ is a may-transition relation, and $\longrightarrow_{\Box S} \subseteq states_S \times act_S \times states_S$ is a must-transition relation. Like previously we only consider syntactically consistent modal I/O automata here, so $\longrightarrow_\Box \subseteq \longrightarrow_\Diamond$.*

The composition for modal I/O automata combines both the modal aspects and the communications aspects. Two modal I/O automata S_1,S_2 are *composeable* iff their actions only overlap on complementary types: $(in_{S_1} \cup int_{S_1}) \cap (in_{S_2} \cup int_{S_2}) = \emptyset$ and $(out_{S_1} \cup int_{S_1}) \cap (out_{S_2} \cup int_{S_2}) = \emptyset$. The composition $S_1 \otimes S_2$ gives rise to a modal I/O automaton S such that $states_S = states_{S_1} \times states_{S_2}$, $start_S = (start_{S_1}, start_{S_2})$, $in_S = (in_{S_1} \setminus out_{S_2}) \cup (in_{S_2} \setminus out_{S_1})$, $out_S = (out_{S_1} \setminus in_{S_2}) \cup (out_{S_2} \setminus in_{S_1})$, $int_S = int_{S_1} \cup int_{S_2} \cup (in_{S_1} \cap out_{S_2}) \cup (out_{S_1} \cap in_{S_2})$. The transition relations are given by the following rules (see Fig. 2 for an example):

$$\frac{s_1 \xrightarrow{a!}_\gamma s_1' \qquad s_2 \xrightarrow{a?}_\gamma s_2'}{s_1 \otimes s_2 \xrightarrow{a}_\gamma s_1' \otimes s_2'} \ \gamma \in \{\Box, \Diamond\} \qquad \frac{s_1 \xrightarrow{a?}_\gamma s_1' \qquad s_2 \xrightarrow{a!}_\gamma s_2'}{s_1 \otimes s_2 \xrightarrow{a}_\gamma s_1' \otimes s_2'} \ \gamma \in \{\Box, \Diamond\}$$

$$\frac{s_1 \xrightarrow{a}_\gamma s_1' \qquad a \notin act_{S_2}}{s_1 \otimes s_2 \xrightarrow{a}_\gamma s_1' \otimes s_2} \ \gamma \in \{\Box, \Diamond\} \qquad \frac{s_2 \xrightarrow{a}_\gamma s_2' \qquad a \notin act_{S_1}}{s_1 \otimes s_2 \xrightarrow{a}_\gamma s_1 \otimes s_2'} \ \gamma \in \{\Box, \Diamond\}$$

For technical reasons (efficiency and simplicity) we always assume that unreachable states are removed after computing a composition (both here and in later sections). The following theorem is a simple corollary from the general fact that the modal refinement is a precongruence [15,16]:

Theorem 8. *Modal refinement is a precongruence with respect to the above composition operator: for any four modal I/O automata T_1, T_2, S_1, S_2 such that $T_1 \leq_\mathrm{m} S_1$ and $T_2 \leq_\mathrm{m} S_2$ it holds that $T_1 \otimes T_2 \leq_\mathrm{m} S_1 \otimes S_2$.*

The composition operator ($\otimes$) defined above corresponds to a usual composition of software (hardware) *components*. Whenever we use it below we mean an unrestricted connection of components, which does not preclude deadlocks or other kinds of errors. We shall soon introduce two seemingly similar composition operators, ($|$) and ($\cdot$) having a very different use. In fact they are algorithms synthesizing *specifications* of how a result of simple composition ($\otimes$) should be used in order to guarantee the absence of certain errors.

5 A Modal Interface Theory

Interface theories support component based development. The aim is to specify component interfaces and from these interfaces to derive the interfaces of composite components. The novel aspect of the interface theory presented here is that the components can specify both required and allowed behavior, consequently it is suitable for expressing liveness properties.

In our specific interface theory an interface is given by a modal I/O automaton. A given interface specifies a set of potential implementations (concrete implementations have identical transition relations $\longrightarrow_\Diamond = \longrightarrow_\Box$). The goal of our interface theory is to be able to use interface descriptions to describe legal implementations of components in a component based system. The implementation relation, the relation that specifies which implementations conform to a given interface description is modal refinement $\leq_{\mathrm{m}}$. From the interface descriptions of two components it should be possible to derive the interface of the combined component. This is done without knowing more about the implementations, than the fact that they conform to their individual interface specification.

The result of composing two interfaces is a subset of the result of composing two modal I/O automata, in which all possible internally controllable paths leading to error states are removed. An *error state* is a state in which one component can output something that the other component might be unable to receive:

$$err^{\mathrm{i}}_{S_1,S_2} = \{(s_1,s_2) \in states_{S_1 \otimes S_2} \mid \text{there exists } a \in int_{S_1 \otimes S_2} \text{ and states } s_1',\ s_2' \text{ such that } (s_1 \xrightarrow{a!}{}^{S_1}_{\Diamond}\ s_1' \text{ and } s_2 \not\xrightarrow{a?}{}^{S_2}_{\Box}) \text{ or } (s_2 \xrightarrow{a!}{}^{S_2}_{\Diamond}\ s_2' \text{ and } s_1 \not\xrightarrow{a?}{}^{S_1}_{\Box})\} \quad (2)$$

State 22 on Fig. 2 is an error state, witnessed by the fail action.

We are now ready to define the set of states of the composition:

$$states_{S_1|S_2} = \bigcap_{n=0}^{\infty} prune^n_{\mathrm{i}}(states_{S_1 \otimes S_2} \setminus err^{\mathrm{i}}_{S_1,S_2}) \ , \quad (3)$$

where $prune_{\mathrm{i}}(S) = \{s \in S \mid \forall s' \forall a \in int_{S_1 \otimes S_2}.\ s \xrightarrow{a}_\Diamond s' \text{ implies } s' \in S\}$, which is a monotonic function that removes, from the set of states S, all those states that in one internally controllable step may reach a state that is not in S.

See Figure 3 (left) for an example of how pruning works. State 22 has been removed as an error state, then state 21 was pruned as an error state can be reached from it by the internally controllable transition log!. Then all transitions involving states 21 and 22 were removed. State 20 remains in the result as the must transition labeled down is externally controllable.

Definition 9 (Composition). *The composition of two interfaces S_1 and S_2 is defined if S_1 and S_2 are composable modal I/O automata and $start_{S_1 \otimes S_2} \in states_{S_1|S_2}$ (see above). The composition results in a modal I/O automaton $S_1|S_2$ such that $S_1|S_2 = (states_{S_1|S_2}, start_{S_1 \otimes S_2}, in_{S_1 \otimes S_2}, out_{S_1 \otimes S_2}, int_{S_1 \otimes S_2}, \longrightarrow_\Diamond^{S_1 \otimes S_2} \cap (states_{S_1|S_2} \times act_{S_1 \otimes S_2} \times states_{S_1|S_2}), \longrightarrow_\Box^{S_1 \otimes S_2} \cap (states_{S_1|S_2} \times act_{S_1 \otimes S_2} \times states_{S_1|S_2}))$.*

Two interfaces are compatible if the set of states resulting from composition, $states_{S_1|S_2}$, contains the initial state $(start_{S_1}, start_{S_2})$.

A desirable property of an interface theory is that components can be implemented independently of each other once the specifications are known. The following theorem formally states that this theory satisfies the property.

Theorem 10 (Independent Implementability). *For any two compatible interfaces S_1, S_2 and for any two implementations I_1, I_2, $I_1 \leq_{\mathrm{m}} S_1$ and $I_2 \leq_{\mathrm{m}} S_2$, it holds that $I_1 \otimes I_2 \leq_{\mathrm{m}} S_1|S_2$.*

This has three implications. First, $I_1 \otimes I_2$ would deliver all the required behavior promised by $S_1|S_2$ as long as it interacts with an environment obeying $S_1|S_2$. Second, $I_1 \otimes I_2$ will not do anything that $S_1|S_2$ would not allow in such an environment. Third, since $S_1|S_2$ does not contain error states then $I_1 \otimes I_2$ will not deadlock.

Theorem 11 (Deadlock Freeness Preservation). *For any two compatible interfaces S_1, S_2, any two implementations I_1, I_2, so $I_1 \leq_{\mathrm{m}} S_1$ and $I_2 \leq_{\mathrm{m}} S_2$, and any interface T compatible with $S_1|S_2$, if $T \otimes (S_1|S_2)$ has no reachable error states then $T \otimes (I_1 \otimes I_2)$ has no reachable error states.*

Finally the composition operator (|) is commutative and associative up to graph isomorphism.

6 A Product Line Theory

In product line development one typically maintains a family of existing *assets* that are composed in a bottom-up fashion in order to build a product. Here we assume that existing assets are sufficient to build the product and no genuinely new programming is required. Assets are organized in small subfamilies, that can be thought of as configurable components. Choosing an asset from a subfamily is a configuration process. We model subfamilies as modal I/O automata, and call them *variability models*, to distinguish them from interfaces. The configuration process amounts to finding a suitable modal refinement of a variability model.

There is a need for a mechanism for composing variability models, to enable reasoning about the products that can be constructed using available assets. As in the interface theory we are interested in computing the legal uses for the composition of two models, without reaching error states. However we weaken the requirement this time: we do not require that *all* possible pairs of implementations give an error free composition, but only that there *exists* a pair of implementations that can avoid errors under a suitable use.

Two variability models are composable if their input, output and hidden actions do not overlap (the general rule for modal I/O automata). Two composable families can be composed, resulting in a description of a higher level component family. The signature of this variability model is found in the same way as for modal I/O automata. The requirement for the description of this more abstract family is that a specification that refines its description can be realized by choosing some concrete implementations from both lower level families involved. So that in effect one can configure the final product by configuring the abstract composed variability model, being sure that the selected configuration can be refined to configurations of each of the smaller components, available in the collection of assets. We give a sufficient condition for a refinement of a variability model to be decomposable.

The ultimate composition closely resembles the composition (|) for interface automata: it uses the regular modal I/O automata composition ($\otimes$) first and then removes error states. However now only internally controllable *required transitions* are pruned, while in the interface theory we had also removed states reachable by *allowed executions* of the same kind. The very existence of allowed internally controlled execution to an error state was considered dangerous in the interface theory—it is not in the product line theory. This is because we are not interested in eliminating errors by all means, but only in making sure that there exist error-free realizations of the specification. For two syntactically composable variability models we define the set of error states, $err^{\mathrm{v}}_{S_1,S_2}$, to be:

$$err^{\mathrm{v}}_{S_1,S_2} = \{(s_1,s_2) \in states_{S_1\otimes S_2} \mid \text{ there exists } a \in int_{S_1\otimes S_2} \text{ and states } s_1', s_2' \text{ such that } (s_1 \xrightarrow{a!}_{\Box} s_1' \text{ and } s_2 \not\xrightarrow{a?}_{\Diamond}) \text{ or } (s_1 \not\xrightarrow{a?}_{\Diamond} \text{ and } s_2 \xrightarrow{a!}_{\Box} s_2')\} \quad (4)$$

In Figure 2 (right) state 22 is still an error state, though for a different reason than previously: in state 22 the *LinkLayer must* be able to produce fail, but the *Client* is *not allowed* to receive it. If a product of two variability models contains an error state it means that there exist configurations of composed assets that cannot safely work together. However, in the same spirit as in the interface theory, we can compute the set of legal uses that guarantee that there *exist* pairs of compatible configurations to interact with them. We remove from the product $S_1 \otimes S_2$ all the states that according to the variability specification *must* be able to reach an error state. If there is no states left then the two variability models are *incompatible*. Otherwise we arrive at a specification of states and transitions among the compatible states that constraint possible legal implementations obtained from these two families. Formally:

$$states_{S_1\cdot S_2} = \bigcap_{n=0}^{\infty} prune^n_{\mathrm{v}}(states_{S_1\otimes S_2} \setminus err^{\mathrm{v}}_{S_1,S_2}) \ , \quad (5)$$

where $prune_{\mathrm{v}}(S) = \{s \in S \mid \forall s'.\, \forall a \in int_{S_1\otimes S_2} \cup out_{S_1\otimes S_2}.\, s \xrightarrow{a}_{\Box} s' \text{ and } s' \in S\}$. We compute the two transition relations for the composition, by projecting the transition relations of the parallel composition $S_1 \otimes S_2$ onto the new set of states:

$$\longrightarrow^{S_1\cdot S_2}_{\Diamond} = \longrightarrow^{S_1\otimes S_2}_{\Diamond} \cap (states_{S_1\cdot S_2} \times act_{S_1\otimes S_2} \times states_{S_1\cdot S_2}) \quad (6)$$

$$\longrightarrow^{S_1\cdot S_2}_{\Box} = \longrightarrow^{S_1\otimes S_2}_{\Box} \cap (states_{S_1\cdot S_2} \times act_{S_1\otimes S_2} \times states_{S_1\cdot S_2}) \ . \quad (7)$$

Finally we can state the complete result of the composition: a modal I/O automaton $S_1 \cdot S_2$ such that $S_1 \cdot S_2 = (states_{S_1\cdot S_2}, (start_{S_1}, start_{S_2}), in_{S_1\otimes S_2}, out_{S_1\otimes S_2}, int_{S_1\otimes S_2}, \longrightarrow^{S_1\cdot S_2}_{\Diamond}, \longrightarrow^{S_1\cdot S_2}_{\Box})$ and all the components are defined above.

Definition 12. *Two variability models are compatible if they are composable and their composition is nonempty.*

It turns out that *observationally consistent* refinements of compositions of variability models are realizable with existing assets. We define observational consistency for states of a single automaton. Let $t \xrightarrow{A}_{\Box}{}^* t'$ mean that t' is reachable

from t via a possible empty sequence of required transitions labeled by possibly different actions from a set A.

Definition 13. *Let T be a modal automaton and let $A \subseteq act_T$ be a set of actions. A relation $C \subseteq states_T \times states_T$ is an observational consistency relation with respect to A if for any pair of states $(t_1, t_2) \in C$ the following two properties hold:*

1. $\forall t_1'.$ *if* $t_1 \xrightarrow{A}{}^*_{\Box} t_1'$ *then* $\forall a \notin A. \forall t_1''. t_1' \xrightarrow{a}_{\Box} t_1''$ *implies* $\exists t_2'. t_2 \xrightarrow{a}_{\Diamond} t_2' \wedge (t_1'', t_2') \in C$.
2. $\forall t_2'.$ *if* $t_2 \xrightarrow{A}{}^*_{\Box} t_2'$ *then* $\forall a \notin A. \forall t_2''. t_2' \xrightarrow{a}_{\Box} t_2''$ *implies* $\exists t_1'. t_1 \xrightarrow{a}_{\Diamond} t_1' \wedge (t_1', t_2'') \in C$.

Two states are observationally consistent if there exists an observational consistency relation relating them. A set of states is said to be observationally consistent with respect to A if all possible pairs of states from the set are observationally consistent with respect to A. An automaton T is observationally consistent with respect to A iff the set $\{start_T\}$ is an observationally consistent set.

The following theorem states the existence of decomposition formally:

Theorem 14 (Decomposability). *Let T_1, T_2 be deterministic composable variability models, and S be a configuration (a deterministic variability model itself) such that $S \leq_{\mathrm{m}} T_1 \cdot T_2$, and T_1, S are observationally consistent with respect to $act_{T_1} \setminus act_{T_2}$ and T_2, S are observationally consistent with respect to $act_{T_2} \setminus act_{T_1}$. Then there exist S_1 and S_2 such that $S_1 \leq_{\mathrm{m}} T_1$ and $S_2 \leq_{\mathrm{m}} T_2$ and $S_1 \otimes S_2 \leq_{\mathrm{m}} S$.*

A version of the theorem, not requiring observational consistency, does not hold, which can be demonstrated with a counter-example, not included here.

An important corollary is that the decomposition can be carried over down to precise configurations: if a concrete configuration of a product is required, then there exist concrete configurations of assets to realize it. The question whether a specification is realizable with given assets is reduced to establishing observational consistency and a modal refinement between the postulated requirement and the variability model. Consequently the abstract variability model can be communicated to configuration engineers and used to configure final products.

Let us close our discussion with a statement that the $(\cdot)$ operator is general enough to describe all implementations safely realizable with existing assets.

Theorem 15 (Completeness). *For any two compatible variability models T_1, T_2 and any two compatible concrete implementation specifications I_1, I_2, where $I_1 \leq_{\mathrm{m}} T_1$ and $I_2 \leq_{\mathrm{m}} T_2$ it holds that $I_1 \cdot I_2 \leq_{\mathrm{m}} T_1 \cdot T_2$.*

7 Conclusion and Future Work

We have investigated the relation between alternating simulation as used in interface automata and observational modal refinement, concluding that former is a case of the latter. We have argued that the strength of the game theoretic

approach to interface theories does not lie in alternating refinement itself, but in the labeling of transitions with control information; in partitioning the actions into internally and externally controllable. We have extended modal transition systems with this information and demonstrated that in this way interface theories tracking liveness properties, can be built. Finally we have presented a product line theory describing variability in behavior of component families.

In the future we would like to extend the product line theory of Section 6 to a full featured theory based on observational modal refinement and study its properties in depth. Also it appears interesting to investigate the relation between the general notion of alternating refinement [8] and (modal) transition systems, lifting the restrictions accepted in Section 3 after the interface automata model.

References

1. Alfaro, L., Henzinger, T.A.: Interface automata. In: Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE), Vienna, Austria, ACM Press (2001) 109–120
2. Larsen, K.G., Thomsen, B.: A modal process logic. In: LICS, IEEE Computer Society (1988)
3. Chakabarti, A., de Alfaro, L., Henzinger, T.A., Stoelinga, M.I.A.: Resource interfaces. In Alur, R., Lee, I., eds.: EMSOFT 03: 3rd Intl. Workshop on Embedded Software. LNCS, Springer (2003)
4. Alfaro, L., Henzinger, T., Stoelinga, M.I.A.: Timed interfaces. In Sangiovanni-Vincentelli, A., Sifakis, J., eds.: EMSOFT 02: 2nd Intl. Workshop on Embedded Software. LNCS, Springer (2002)
5. Larsen, K.G., Nyman, U., Wąsowski, A.: Interface input/output automata. In Misra, J., Nipkow, T., Sekerinski, E., eds.: 14th International Symposium on Formal Methods (FM) Hamilton, Canada, August 21–27, 2006 Proceedings. Volume 4085 of LNCS., Springer (2006) 82–97
6. Černá, I., Vařeková, P., Zimmerová, B.: Component substitutability via equivalencies of component-interaction automata. In: FACS'06. (2006) 115–130 To be published in ENTCS.
7. Hermanns, H., Rehof, J., Stoelinga, M.I.A., eds.: Workshop Procedings FIT 2005: Foundations of Interface Technologies. ENTCS, Elsevier Science Publishers (2005)
8. Alur, R., Henzinger, T.A., Kupferman, O., Vardi, M.: Alternating refinement relations. In Sangiorgi, D., de Simone, R., eds.: Proceedings of the Ninth International Conference on Concurrency Theory (CONCUR'98). Volume 1466 of LNCS., Springer (1998) 163–178
9. Alfaro, L., Henzinger, T.A.: Interface-based design. In: In Engineering Theories of Software Intensive Systems, Marktoberdorf Summer School, Kluwer Academic Publishers (2004)
10. Carrez, C., Fantechi, A., Najm, E.: Assembling components with behavioral contracts. Annales del Télécommunications **60** (2005)
11. Parnas, D.L.: On the design and development of program families. IEEE Transactions on Software Engineering **Vol. SE-2** (1976) 1–9
12. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)

13. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering—Foundations, Principles, and Techniques. Springer (2005)
14. Larsen, K.G., Larsen, U., Wąsowski, A.: Color-blind specifications for transformations of reactive synchronous programs. In Cerioli, M., ed.: FASE, Edinburgh, April 2005. LNCS, Springer (2005)
15. Boudol, G., Larsen, K.G.: Graphical versus logical specifications. In Arnold, A., ed.: CAAP. Volume 431 of Lecture Notes in Computer Science., Springer (1990) 57–71
16. Larsen, K.G.: Modal specifications. In Sifakis, J., ed.: Automatic Verification Methods for Finite State Systems. Volume 407 of Lecture Notes in Computer Science., Springer (1989) 232–246