Revisiting Compatibility of Input-Output Modal Transition Systems[⋆]

Ivo Krka[1], Nicolás D'Ippolito[2,3], Nenad Medvidović[4], and Sebastián Uchitel[2,3]

[1] Google Inc, Zürich, Switzerland
[2] Computing Department, Imperial College London, London, UK
[3] Departamento de Computatión, FCEyN, Universidad de Buenos Aires, Argentina
[4] University of Southern California, Los Angeles, CA, USA

Abstract. Modern software systems are typically built of components that communicate through their external interfaces. The external behavior of a component can be effectively described using finite state automata-based formalisms. Such component models can then used for varied analyses. For example, interface automata, which model the behavior of components in terms of component states and transitions between them, can be used to check whether the resulting system is compatible. By contrast, partial-behavior modeling formalisms, such as modal transition systems, can be used to capture and then verify properties of sets of prospective component implementations that satisfy an incomplete requirements specification. In this paper, we study how pairwise compatibility should be defined for partial-behavior models. To this end, we describe the limitations of the existing compatibility definitions, propose a set of novel compatibility notions for modal interface automata, and propose efficient, correct, and complete compatibility checking procedures.

1 Introduction

Modern software systems are typically built of components that communicate through their external interfaces. A component's behavior can be specified using finite state automata formalisms (e.g., Labeled Transition Systems [8] and Statecharts [7]). The basic formalism, Labeled Transition Systems (LTS), describes the behavior of a component in terms of component states and labeled transitions between them. Interface Automata (IA) [1] extend LTS to model information related to interface operation *controllability* —distinguishing between input, output, and internal actions— and to check whether the interfaces of two components are *semantically compatible*.

Component's behavior is often incrementally and iteratively refined and elaborated as the requirements progressively become more complete. *Partial-behavior* modeling formalisms (e.g., Modal Transition Systems (MTS) [14]) distinguish between required behaviors, prohibited behaviors, and behaviors that are currently unknown as either

⋆ This work was partially supported by grants ERC PBM-FIMBSE, ANPCYT PICT 2012-0724, UBACYT W0813, ANPCYT PICT 2011-1774, UBACYT F075, CONICET PIP 11220110100596CO, MEALS 295261, and U.S. NSF awards 0905665, 1117593, and 1218115. The work has been done while Ivo Krka was a PhD candidate at the University of Southern California.

required or prohibited. Hence, such models can accurately capture the inherently partial system requirements and serve as a foundation for iterative practices that involve eliciting new requirements that prohibit or require some of the previously unknown behaviors [4, 5, 9, 11, 12, 19–21]). Partial behavior models come equipped with a notion of *refinement* which formalizes the process of incorporating new requirements into the partial specification. For example, a partial-behavior model of a product under development is refined by selecting or discarding a specific feature. The final result of the refinement process is a model without unknown behavior (e.g., an interface automaton) that we refer to as an *implementation*.

At the implementation level, two components, represented as IAs, are compatible if the output actions of one component are not blocked by a lack of matching input actions in the other component. To enable continuous interface compatibility checking when a specification is partial and iteratively refined, several *modal interface automata* formalisms have been proposed [2, 13, 18]. Intuitively, determining the compatibility of partially specified components should characterize "how compatible" those components' implementations are [10]. For example, at one extreme, any selection of implementations results in an error-free system (i.e., highly compatible partial specifications). At the other extreme, only a very careful selection of implementations results in an error-free system (i.e., conditionally compatible specifications). Therefore, the compatibility of partial specifications directly affects how independently engineers can specify the requirements for the different subsystems and components.

While promising, the prior work on modal interface automata is limited in terms of the considered compatibility notions, as elaborated in our prior study [10]. In particular, the existing work implicitly considers only the above two compatibility extremes: either *all* pairs of implementations are compatible vs. *at least one* pair of implementations is compatible. A richer and finer-grained spectrum of compatibility notions is needed so that engineers can determine that *specific subsets* of the modal interface automata implementations are compatible. In turn, such richer compatibility notions would inform the subsequent specification refinement processes and make them more flexible and loosely coupled. For example, consider the case when every implementation of one partially specified component has a compatible counterpart in the other component's set of possible implementations. The first component can then be refined independently, followed by careful refinement of the other component (we refer to this case as *Implementation Compatibility*).

In this paper, we revisit compatibility of Input-Output Modal Transition Systems (IO MTS), i.e., MTS extended with input and output information. We define a range of IO MTS compatibility notions *semantically*, based on the observation that IO MTS are used to express sets of implementations. In contrast, previous work on such specifications provided only syntactic definitions of compatibility. Our work lets an engineer determine whether some, all, or no implementations from one component's implementation set are compatible with some, all, or no implementations from another component's implementation set. Given that the implementation sets may be infinite, for each compatibility notion we propose a correct and complete procedure that, for two IO MTSs with a finite set of transitions, efficiently checks their compatibility by checking the compatibility of specially constructed implementations. While we define compati-

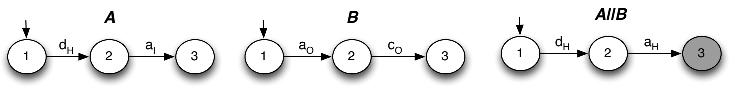

Fig. 1. Example interface automata and modal I/O automata for illustration of Compatibility.

bility in a pair-wise fashion, the definitions of compatibility can be trivially extended to N-way relationships between the system components' implementation sets.

The main contributions of this paper are: (1) general, *semantics-based* definitions of *four compatibility notions* for IO MTS; (2) a discussion of the development processes that are enabled by each compatibility notion; (3) novel concepts of *the least constraining implementation* and *the most constraining implementation* of an IO MTS; and (4) a set of correct, complete, and efficient procedures for checking compatibility of two IO MTS based on their least/most constraining implementations.

The next section provides the foundations of our work. Section 3 defines a set of four novel compatibility notions. Section 4 proposes a suite of procedures for checking IO MTS compatibility. Finally, Section 5 discusses the implications of the new compatibility notions and concludes the paper.

2 Background

To understand how we modify the notions of compatibility for modal interface specifications, it is necessary to first introduce the formalisms for specifying complete component interfaces and partial component behaviors, and then to introduce how compatibility is currently defined for such specifications.

2.1 Transition Systems

A labeled transition system [17] is an FSA-based formalism used to model required behavior of a software component as a set of component states and labeled transitions between them. De Alfaro's interface automata (IA) [1] are an extension of LTS that distinguishes between input, output, and internal actions. The distinction between these different types of actions enables the detection of communication mismatches (i.e., incompatibilities) when the automata are composed.

Definition 1 (IA). *An interface automaton IA is a tuple (S, A^I, A^O, A^H, Δ, s_0), where S is a set of states, A^I, A^O, A^H are alphabets of input, output, and internal actions, $\Delta \subseteq (S \times A^I \cup A^O \cup A^H \times S)$ is the transition relation, and s_0 is the initial state.*

We use the notation $s \xrightarrow{\ell_\omega} s'$ for a required transition from s to s' labeled with ℓ, $\omega \in \{I, O, H\}$ denotes input, output and internal transitions respectively. We may refer to states in an IA A using dot notation, e.g. $A.s_1$ refers to the state s_1 of A.

Two IAs M and N are *composable* if they do not share any internal, input or output actions (i.e., $A^H_M \cap A_N = \emptyset$, $A^I_M \cap A^I_N = \emptyset$, $A^O_M \cap A^O_N = \emptyset$ and $A^H_N \cap A_M = \emptyset$). Models A and B, in Figure 1, are examples of composable IAs.

Interface automata have a composition operator [1]; for brevity, we only define the more general composition of IO MTS. The composition of IAs M and N is defined as a restriction on the synchronous product automaton $M \otimes N$, which coincides with the composition of I/O automata [16].

Definition 2. (Product) *Given M =(S_M, A^I_M, A^O_M, A^H_M, Δ_M, m_0) and N =(S_N, A^I_N, A^O_N, A^H_N, Δ_N, n_0) composable interface automata, their product is the interface automaton* $M \otimes N = \langle S_M \times S_N, A^I_{M\otimes N}, A^O_{M\otimes N}, A^H_{M\otimes N}, \Delta_{M\otimes N}, (m_0, n_0)\rangle$ *where*

$$\begin{aligned} A^I_{M\otimes N} &= (A^I_M \cup A^O_N) \setminus A_M \cup A_N \\ A^O_{M\otimes N} &= (A^O_M \cup A^O_N) \setminus shared(M,N) \\ A^H_{M\otimes N} &= (A^H_M \cup A^H_N) \cup shared(M,N) \end{aligned}$$

The transition relation is defined as follows:

$$\begin{aligned} Delta_{M\otimes N} = &\{((m,n),\ell,(m',n))|(m,\ell,m') \in \Delta_M \wedge \ell \notin shared(M,N)\} \\ &\cup \{((m,n),\ell,(m,n'))|(n,\ell,n') \in \Delta_N \wedge \ell \notin shared(M,N)\} \\ &\cup \{((m,n),\ell,(m',n'))|(m,\ell,m') \in \Delta_M \wedge (m,\ell,m') \in \Delta_M \wedge \ell \in shared(M,N)\} \end{aligned}$$

Let $shared(M,N) = A_M \cap A_N$.

A condition for interface automata composition ($||$) is that an input event of one automaton can only be an output event of another automaton. Furthermore, composing an input action in one automaton with a matching output action in the other automaton produces an internal action in the composition. For example, the interface automata $A||B$ in Figure 1 that represents the composition of A and B has internal transition over a that is the result of A and B synchronizing on a ($A.s_2 \xrightarrow{a_I} A.s_3$ and $B.s_1 \xrightarrow{a_O} B.s_2$).

In order to model uncertain aspects, or currently missing and underspecified aspects, of a system's behavior, Larsen and Thomsen proposed modal transition systems (MTS) [14]. MTS generalizes LTS with *maybe* transitions that are currently neither explicitly required nor prohibited, in addition to the *required* transitions found in LTS. The disjoint sets of required and maybe transitions comprise a set of potential transitions. Intuitively, an MTS describes a set of possible LTSs by describing an upper bound and a lower bound of allowed behaviors from every state.

Definition 3 (MTS). *A modal transition system M is a tuple (S, A, Δ^r, Δ^p, s_0), where S is the set of states, A is the action alphabet, $\Delta^r \subseteq S \times A \times S$ is the required transition relation, $\Delta^p \subseteq S \times A \times S$ is the potential transition relation, $\Delta^r \subseteq \Delta^p$, and s_0 is the initial state.*

As more information about the desired system behavior becomes available, some of the maybe behavior in an MTS may become required, while other maybe behavior may become prohibited. In this context, it is necessary to ensure that the revised partial models and the eventually obtained final model (referred to as an *implementation*) conform to the initially developed partial model.

Definition 4. (Refinement) *Let* $M = (S_M, A, \Delta^r_M, \Delta^p_M, m_0)$ *and* $N = (S_N, A, \Delta^r_N, \Delta^p_N, n_0)$ *be two MTSs. Relation* $R \subseteq S_M \times S_N$ *is a* refinement *between M and N if the following holds for every* $\ell \in A$ *and every* $(s,t) \in R$.

- *If $(m,\ell,m') \in \Delta^r_M$ then there is n' such that $(n,\ell,n') \in \Delta^r_N$ and $(m',n') \in R$.*
- *If $(n,\ell,n') \in \Delta^p_N$ then there is m' such that $(m,\ell,m') \in \Delta^p_M$ and $(m',n') \in R$.*

We say that N refines M if there is a refinement relation R between M and N such that $(m_0,n_0) \in R$, denoted $M \preceq N$.

Intuitively, N refines M if every required transition of M exists in N and every possible transition in N is possible also in M. An LTS can be viewed as an MTS where $\Delta^p = \Delta^r$. LTSs that refine an MTS M are complete descriptions of the system behavior and are thus called *implementations* of M, denoted $Impls(M)$. An MTS N is a refinement of an MTS M iff the implementation set of N is a subset of M's implementations.

To model communication control in the presence of partially known requirements, formalisms such as Modal I/O automata [2,13], Modal Interfaces [18], and Modal Interface Automata [10] have been proposed. In essence, these formalisms merge MTS and IA formalisms. Since MTS is the most widely used partial-behavior formalism, in this paper we refer to this merge as an Input-Output Modal Transition System (IO MTS). Intuitively, an IO MTS represents a set of IA implementations.

Definition 5 (Input-Output Modal Transition Systems). *An input-output modal transition system IO is a tuple $(S, A^I, A^O, A^H, \Delta^r, \Delta^p, s_0)$, where S is a set of states, A^I, A^O, A^H are alphabets of input, output, and internal actions respectively, $\Delta^r \subseteq S \times (A^I \cup A^O \cup A^H) \times S$ is the required transition relation, $\Delta^p \subseteq S \times (A^I \cup A^O \cup A^H) \times S$ is the potential transition relation ($\Delta^r \subseteq \Delta^p$), and s_0 is the initial state.*

We refer to transitions in $\Delta^p \setminus \Delta^r$ as "maybe" transitions to distinguish them from required ones (those in Δ^r). Maybe transitions are denoted by suffixing the transition label with "?" (e.g., $s \xrightarrow{\ell_I?} s'$). For a given IO MTS M we denote $M.\Delta^\delta_\alpha$ the set of transitions in Δ^α over actions in δ, for instance, $M.\Delta^r_I$ is the set of required transitions over internal actions of M.

For example, consider the IO MTS A from Figure 2. The maybe transition $A.s_1 \xrightarrow{a_I?} A.s_2$ implies that a decision on whether a will be implemented or not in state $A.s_1$ has not been made yet. By contrast, the required transition $B.s_1 \xrightarrow{b_H} B.s_1$ in the B IO MTS of Figure 2 implies that b must be present in every implementation of B.

2.2 Interface Compatibility

As stated above, the composition of interface automata may involve communication errors; the definition of interface automata compatibility [1] implies that two automata are compatible if in their composition errors can be avoided.

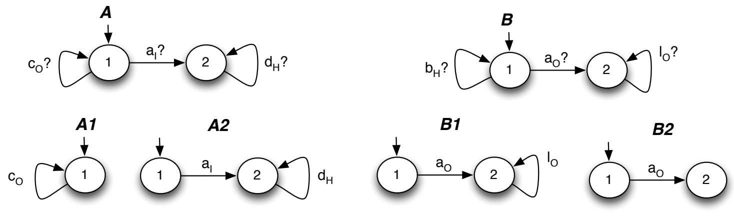

Fig. 2. Conditionally Compatible Models

Definition 6 (IA Error State). *Let IA_1 and IA_2 be interface automata. A state $P.v = \langle IA_1.s, IA_2.t\rangle$ in the interface automaton $P = IA_1||IA_2$ is an error state iff for some $l \in (IA_1.A_O \cap IA_2.A_I) \cup (IA_1.A_I \cap IA_2.A_O)$:*

1. *$(\exists IA_1.s \xrightarrow{\ell_O} IA_1.s') \wedge (\neg\exists IA_2.t \xrightarrow{\ell_I} IA_2.t')$, or*
2. *$(\neg\exists IA_1.s \xrightarrow{\ell_I} IA_1.s') \wedge (\exists IA_2.t \xrightarrow{\ell_O} IA_2.t')$.*

We use $Err(IA_1, IA_2)$ to denote the set of error states.

Definition 7 (IA Compatibility). *Two interface automata IA_1 and IA_2 are compatible if they are nonempty, composable, and there exists an IA E such that no state in $Err(IA_1, IA_2) \times E.s$ is reachable in $(IA_1||IA_2)||E$.*

The IA E in the above definition is referred to as a *Legal Environment* for $(IA_1||IA_2)$.

Informally, a composite state is an error state when, for the composed component states, an output transition in one automaton does not have a matching input transition in the other automaton. Two IAs are considered compatible if their composition can operate error-free in some environments (an environment is an external entity, represented as IA, that uses the system). For example, the composite state $(A\|B).s_3$ of $A\|B$ from Figure 1 is an error state because B can generate c_O from state $B.s_2$ in B, while A does not accept c_I in state $A.s_3$.

Larsen and Thomsen [13], as well as subsequent work by other authors [2, 18], attempt to adapt the definition of compatibility from IA to IO MTS. To this end, they propose different types of error states based on the potential mismatches of output transitions in one IO MTS and input transition in the other IO MTS.

Definition 8 (IO MTS Potential Error State). *A state (s_1, s_2) is a potential error state if there exists $\ell \in {A^H}_{s_1\|s_2}$ such that ($s_1 \xrightarrow{\ell_O?} s_1'$ and $s_2 \not\xrightarrow{\ell_I}$) or ($s_1 \not\xrightarrow{\ell_I}$ and $s_2 \xrightarrow{\ell_O?} s_2'$).*

Definition 9 (IO MTS Mandatory Error State). *A state (s_1, s_2) is a mandatory error state if there exists $\ell \in {A^H}_{s_1\|s_2}$ such that ($s_1 \xrightarrow{\ell_O} s_1'$ and $s_2 \not\xrightarrow{\ell_I?}$) or ($s_1 \not\xrightarrow{\ell_I?}$ and $s_2 \xrightarrow{\ell_O} s_2'$).*

The potential error state implies that a composite state may become an IA error state if refined in a particular manner – e.g., by implementing an output transition from s_1 that is not enabled in s_2. In contrast, a mandatory error state implies that a composite state will be an IA error state if it is reachable in the eventual implementation.

Based on the error state definitions, Larsen et al. [13] define two notions of compatibility for IO MTS. The first definition states that two IO MTSs are compatible if a potential error state is not reachable from the initial state via potential internal actions of the composition. This implies that, no matter the refinement choices, an error-avoiding environment can be built. In other words, all implementations of two compatible IO MTSs will be compatible (Independent Implementability property in [13]).

Larsen's second definition of compatibility states that two IO MTSs are compatible if a mandatory error state is not reachable from the initial state via a set of required internal and output actions of the composition. Under this notion, two compatible IO MTSs can be refined into a pair of compatible implementations (within an appropriate environment). However, this definition does not suggest how the refinement process may proceed, other than by treating the system as a monolithic entity (i.e., every refinement of one IO MTS needs to be synchronized and consistent with the refinements of the other IO MTS).

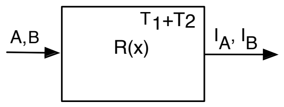

Fig. 3. Fully Coupled Refinement Process

3 Semantically Defining Compatibility

The *limitations* of the existing IO MTS definitions, which we address in the remainder of this paper, are twofold. First, they define compatibility using the syntactic definitions of error states although IO MTS are used to represent sets of implementations, and a more intuitive way of defining IO MTS compatibility would be through compatibility of the possible pairs of implementations. In turn, the syntactic definition may not be applicable more widely, to any type of partial-behavior model. Second, these definitions were developed to solve specific problems (e.g., determining whether there is a compatible product in a product line [13]), and do not explore the full space of possible compatibility notions. We have developed compatibility notions that consider how the implementation sets of the partial specifications relate in terms of their compatibility (one-to-one, one-to-many, many-to-one, or many-to-many). Note that, while we define compatibility notions in the context of IO MTS, they apply generally to partial-behavior models thus serving as a potential common vocabulary for the research community.

The different compatibility notions induce a set of refinement processes they permit. We define these processes and depict them using box-and-line diagrams: A box represents a refinement process that, given an IO MTS to refine among other inputs, produces an implementation of the input IO MTS. The labels Ti inside the boxes denote the independent development teams responsible for the particular process. The arrows denote the information flow between the refinement processes, while the arrow labels specify the information being carried. For example, Figure 3 depicts a situation where two teams, $T1$ and $T2$, are refining a pair of partial specifications, A and B. The incoming arrow to the refinement process, carried out through mutual effort of the two teams ($T1+T2$), indicates that the the teams need to constantly work in concert in order to proceed with the refinement. The outputs I_A and I_B correspond to the implementations obtained by $T1+T2$ after refining A and B.

3.1 Conditional Compatibility

The minimal requirement to consider two IO MTSs A and B compatible is to have at least one compatible system implementation – i.e., a compatible pair (A_i, B_j) of their implementations. Otherwise, no matter which refinement choices are made on A and B, it is impossible to arrive at an error-free system. This weakest compatibility notion has been discussed and syntactically defined in the context of product lines [13], and we refer to it as *Conditional Compatibility*.

Definition 10 (Conditional Compatibility). *Given A and B IO MTSs, we say that A and B are Conditionally Compatible if there exist two implementations $I_A \in Impls(A)$ and $I_B \in Impls(B)$ such that I_A and I_B are compatible.*

In Figure 2, we depict two partial specifications with a compatible pair of implementations (A_2, B_2). While refining A and B into more defined partial models, it is necessary to ensure that the resulting partial specifications contain at least some compatible implementations. For example, if A is refined into A_1 then the only allowed intermediate refinements of B are those that contain B_1 in the implementation set, i.e., those that enable the output transition on a and disable the output transition on l.

The above example suggests that the refinement choices made on the different specifications need to be carefully synchronized: each intermediate refinement of component A needs to be immediately communicated in order to proceed with legal refinement of the other component B, and vice versa. This observation generalizes into a coupled refinement process depicted in Figure 3, where the teams $T1$ and $T2$ are supposed to be in charge of refining the specifications A and B, respectively. Although these are ideally separate teams, conditional compatibility of partial specifications leads to their full coupling — every refinement choice on either A or B strongly impacts the future legal refinements and needs to be carefully negotiated and planned.

3.2 Specification Compatibility

Conditionally compatible specifications entail the weakest requirement for IO MTS compatibility that induces an undesirably highly coupled refinement process. Decreasing this coupling would imply that at least one specification can be refined relatively independently. To this end, we propose two novel, stronger compatibility notions — *Specification Compatibility* (described in this section) and *Implementation Compatibility* (described in the next section).

Specification Compatibility, formalized below, relies on the existence of a subset of one component's implementations that are compatible with every implementation of the other component's partial specification.

Definition 11 (Specification Compatibility). *Given A and B two IO MTSs, we say that A and B are Specification Compatible if there exist $I_A \in Impls(A)$ such that for all $I_B \in Impls(B)$ it holds that I_A and I_B are compatible.*

Consider the two Specification Compatible models A and B in Figure 4. The implementation A_1 of A is compatible with all implementations of B. Hence, as long as A is

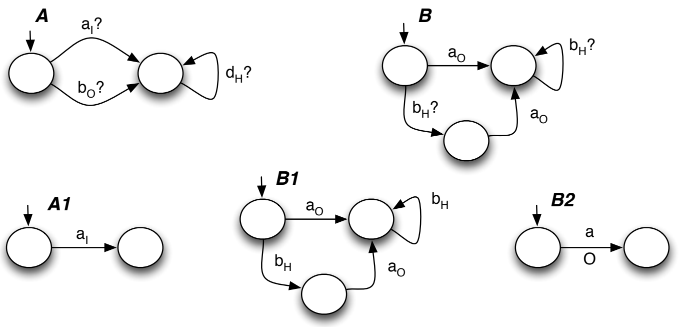

Fig. 4. Specification Compatible Models

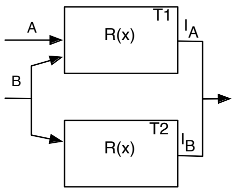

Fig. 5. Specification Driven Refinement

refined into one of those implementations that are consistent with all implementations of B (e.g., A_1), the system is guaranteed to have no error states.

Thus, as depicted in Figure 5, Specification Compatibility induces a process in which the specifications can be refined in parallel. In order to guarantee compatible implementations team $T1$ requires the knowledge of the partial specification B, in addition to its own specification A. By contrast, team $T2$ only requires the specification B and can refine B fully independently, under the condition that team $T1$ respects the partial specification B as a contract that restricts the allowed refinements.

3.3 Implementation Compatibility

As indicated above, Implementation Compatibility implies a less restrictive compatibility notion than Conditional Compatibility that reduces the coupling between the allowed refinement processes of two partial specifications. The relation between the compatibility sets in this case is that every implementation of one IO MTS should have at least one matching pairing in the other implementation set (for the Implementation Compatibility notion of partial-behavior models, the set of matches need not overlap).

Definition 12 (Implementation Compatibility). *Given A and B IO MTSs, we say that A and B are Implementation Compatible if for all $I_A \in Impls(A)$, there exists an $I_B \in Impls(B)$ such that I_A and I_B are compatible.*

For the example depicted in Figure 6, each of the implementations A_1–A_3 of A appears in the compatible set for at least one implementation of B. In particular, A_1 is compatible with B_3, A_2 is compatible with B_2, and A_3 is compatible with B_1. Under

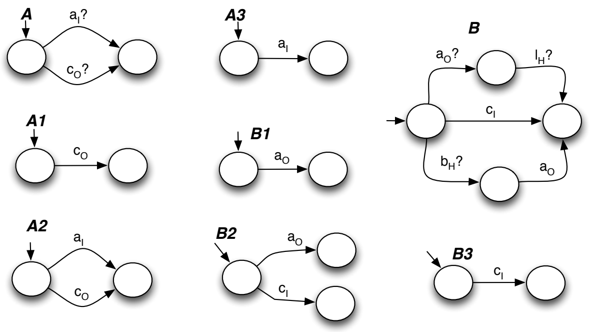

Fig. 6. Implementation Compatible Models

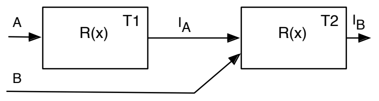

Fig. 7. Implementation Driven Refinement

these conditions, it is guaranteed that whatever implementation of A is chosen, it is possible to find a matching compatible implementation of B.

The implication of Implementation Compatibility is that the first specification, A, can be freely refined without regard for the other specification, B, as long as an appropriate implementation of B is carefully selected afterward. The corresponding process is depicted in Figure 7: the process is sequential as team $T1$ must wait until $T1$ releases an implementation of A. The difference compared to the process for Conditional Compatibility from Figure 7 is that team $T1$ can freely select the refinement choices. These choices in principle stem from the new requirements for component A, while having a guarantee that those new requirements will be consistent with an eventual implementation of B. Hence, Implementation Compatibility is particularly desired in the context of incremental refinement processes where the system is developed one feature at a time (for a large system, a chain of Implementation Compatible IO MTSs would be built).

3.4 Strong Compatibility

The strongest notion of compatibility for a pair of partial specifications A and B is one in which every pair (A_i, B_j) of their implementations is compatible. Consider A and B IO MTSs in Figure 8. As A and B only differ on internal transitions and all their implementations enable the transition on a it follows that A and B are Strong Compatible. This strict notion of compatibility has been used in prior work [2, 13, 18], where it was proposed as the primary notion of compatibility. However, the motivation for our work was that such a notion is overly strict for incrementally developed partial specifications.

Definition 13 (Strong Compatibility). *Given A and B IO MTSs. We say that A and B IO MTSs are Specification Compatible if for all $I_A \in Impls(A)$ and for all $I_B \in Impls(B)$, it holds that I_A and I_B are compatible.*

The direct consequence of having a pair of Strong Compatible specifications is that they can be refined in a fully distributed manner, as depicted in Figure 9. Two teams $T1$ and $T2$ can independently refine their respective specifications, while guaranteeing that the resulting system will operate in an error-free manner. Although achieving Strong Compatibility is desirable due to the consequent parallelism of the refinement process, its importance and prominence is likely to be limited in practice. This is because partial-behavior models are expected to become Strong Compatible only during late stages of the refinement process.

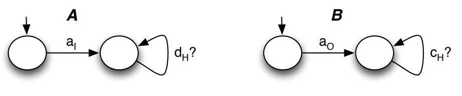

Fig. 8. Strong Compatible Models

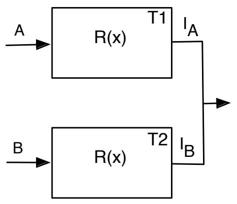

Fig. 9. Fully Distributed Refinement.

4 Checking IO MTS Compatibility

A direct way to check compatibility of two IO MTSs is to check compatibility of their implementations in a pairwise fashion. However, since the implementation sets may be infinite, this is not feasible. Alternatively, it may be possible to separately define an error state for each new compatibility notion. However, it is unclear whether a compact definition of an error state and the associated checking procedure always exist.

The solution we propose is inspired by the concepts of *pessimistic* implementation and *optimistic* implementation of an MTS [20]. The pessimistic implementation is a lower bound of an MTS's behaviors (i.e., no other implementation exhibits less behavior), while the optimistic implementation is an upper bound of an MTS's behaviors (i.e., no other implementation exhibits more behavior). We raise these concepts to IO MTS by defining *the least restrictive implementation* and *the most restrictive implementation* of an IO MTS (Sections 4.1 and 4.2). We then show how these implementations can be used to check, correctly and completely, the compatibility of IO MTS by simply checking the compatibility of their IA composition.

4.1 Least Restrictive Implementation

A component's IA specification describes (1) *assumptions* that the component makes regarding other components' capabilities (via output transitions), (2) *assertions* about how the component progresses internally (via internal transitions), and (3) *guarantees* about the behavior accepted by the component (via input transitions). An implementation set of an IO MTS describes IAs that make a range of assumptions, assertions, and guarantees, depending on which refinement choices were made to arrive at a particular implementation. In this context, the upper bound of an IO MTS interface description would be an IA that makes minimal assumptions and assertions about the output and internal behaviors, while providing maximal guarantees regarding the input behaviors.

Definition 14 (Least Restrictive Implementation). *IA LRI =(S, A^I, A^O, A^H, Δ, s_0) is the least restrictive implementation of an IO MTS M=(S, A^I, A^O, A^H, Δ^r, Δ^p, s_0) with the relation LRI.Δ defined as the union of $M.\Delta^p_I$, $M.\Delta^r_H$, and $M.\Delta^r_O$.*

Informally, the least restrictive implementation of an IO MTS prohibits all the maybe output and maybe internal behaviors of an IO MTS, thus making weaker assumptions and assertions about those behaviors. Similarly, the least restrictive implementation requires all the maybe input behaviors of an IO MTS. A desired "upper bound" property for the least restrictive implementation is to be compatible with every environment that is compatible with at least one implementation of the IO MTS.

Theorem 1 (Upper Bound of Compatibility). *Let IA LRI =(S, A^I, A^O, A^H, Δ, s_0) is the least restrictive implementation of an IO MTS M=(S, A^I, A^O, A^H, Δ^r, Δ^p, s_0). For each other IA I that implements M, if IA E is a legal environment of I then E is also compatible with LRI.*

Proof (By Contradiction). Let IA E be an IA that is compatible with M's implementation I, but is not compatible with *LRI*. This implies that there exists an error state $\langle E.p, LRI.s\rangle$ in which either (1) E generates an output *LRI* cannot accept or (2) *LRI* generates an output that E cannot accept. Note that $LRI.s$ refines a corresponding IO MTS state $M.s$. If the composition $E||I$ has a state $\langle E.p, I.s'\rangle$, where $I.s'$ refines $M.s$ then $\langle E.p, I.s'\rangle$ would also be an error state because, by Definition 14 and for I to be a correct refinement of M [6], $I.s'$ accepts at most as many inputs as *LRI*, and $I.s'$ requires at least as many outputs (thus satisfying condition (1) or (2) above).

In case $E||I$ does not have a state $\langle E.p, I.s'\rangle$ such that $I.s'$ refines $M.s$, consider a sequence of *LRI*'s actions $\langle l_1, \ldots, l_n\rangle$ that are traversed from $\langle E.p_0, LRI.s_0\rangle$ to $\langle E.p, LRI.s\rangle$. Now consider a subsequence $\langle l_1, \ldots, l_j\rangle$ which is supported by I. The next action l_{j+1} in the full sequence cannot be an output or internal action of I because: (a) in case l_{j+1} was required in the matching IO MTS state $M.s_j$, it would be present in both *LRI* and I to satisfy the refinement relation, and (b) in case l_{j+1} was maybe in the matching IO MTS state $M.s_j$, it would also be prohibited in *LRI* per Definition 14. Hence, the action l_{j+1} has to be an input action. This, however, implies that the composite state $\langle E.p_j, I.s'_j\rangle$ is an error state because, in the composite state $\langle E.p_j, LRI.s_j\rangle$, E can generate the output l_{j+1}, which is not accepted in $I.s'_j$. $\square$

4.2 Most Restrictive Implementation

In contrast to the least restrictive implementation, the lower bound of an IO MTS interface description would be an IA that makes maximal assumptions and assertions about the output and internal behaviors, with minimal guarantees on the input behaviors.

Definition 15 (Most Restrictive Implementation). *IA MRI =(S, A^I, A^O, A^H, Δ, s_0) is the most restrictive implementation of an IO MTS M=(S, A^I, A^O, A^H, Δ^r, Δ^p, s_0) with the relation MRI.Δ defined as the union of $M.\Delta^r_I$, $M.\Delta^p_H$, and $M.\Delta^p_O$.*

Informally, the most restrictive implementation of an IO MTS prohibits all the maybe input behaviors of an IO MTS, thus accepting less output behaviors of external components. Similarly, the most restrictive implementation requires all the maybe output and maybe internal behaviors of an IO MTS, thus "forcing" the external components to accept more of its output behaviors. A desired "lower bound" property for the most restrictive implementation is to be compatible with an environment only if every other implementation of the IO MTS is compatible with that environment.

Theorem 2 (Lower Bound of Compatibility). *Let IA MRI =(S, A^I, A^O, A^H, Δ, s_0) be the most restrictive implementation of an IO MTS M=(S, A^I, A^O, A^H, Δ^r, Δ^p, s_0). For each other IA I that implements M, if IA E is a legal environment of MRI then E is also compatible with I.*

Proof (By Contradiction). Let IA E be an IA that is compatible with *MRI*, but not with some other implementation I of M. This implies that there exists an error state $\langle E.p, I.s' \rangle$ in which either (1) E generates an output I cannot accept or (2) I generates an output that E cannot accept. Note that $I.s'$ refines a corresponding IO MTS state $M.s$. If the composition $E||MRI$ has a state $\langle E.p, MRI.s \rangle$, where $MRI.s$ refines $M.s$ then $\langle E.p, MRI.s \rangle$ would also be an error state because, by Definition 15 and for I to be a correct refinement of M [6], $MRI.s$ accepts at most as many inputs as $I.s'$, and $MRI.s$ requires at least as many outputs (thus satisfying condition (1) or (2) above).

In case $E||MRI$ does not have a state $\langle E.p, MRI.s \rangle$ such that $MRI.s$ refines $M.s$, consider a sequence of I's actions $\langle l_1, \ldots, l_n \rangle$ that are traversed from $\langle E.p_0, I.s_0 \rangle$ to $\langle E.p, I.s' \rangle$. Now consider a subsequence $\langle l_1, \ldots, l_j \rangle$ which is supported by *MRI*. The next action l_{j+1} in the full sequence cannot be an input action of *MRI* because: (a) in case l_{j+1} was required in the matching IO MTS state $M.s_j$, it would be present in both *MRI* and I to satisfy the refinement relation, and (b) in case l_{j+1} was maybe in the matching IO MTS state $M.s_j$, it would be prohibited in *MRI* per Definition 15, thus creating an error state as E outputs l_{j+1} in that state. Hence, the action l_{j+1} has to be an output or internal action. However, since *MRI* requires each potential transition on output and internal actions, l_{j+1} would, by construction, exist in $MRI.s_j$. □

4.3 Compatibility Checking Procedure

The least restrictive and most restrictive implementations bound the space of compatible environments for IO MTS implementations. For example, to check whether all implementations of an IO MTS are compatible with an IA, it is sufficient to check the compatibility of the most restrictive implementation with the given IA. Such "bounding" implementations can be used to construct general procedures for checking compatibility of two IO MTSs. For example, Conditional Compatibility requires that at least one pair of implementations of two IO MTSs is error-free. Hence, the intuition is that, at a minimum, their least constraining implementations need to be compatible. In the following definition, we specify how the least constraining implementations and the most constraining implementations of two IO MTSs are used to check IO MTS compatibility. We then prove that the checking procedure is correct and complete for Conditional Compatibility and Specification Compatibility; proofs for other two notions are similar.

Theorem 3 (Checking IO MTS Compatibility). *Let LRI_M and LRI_N be the least restrictive implementations of IO MTS M and N, respectively, and MRI_M and MRI_N be their most restrictive implementations. M and N are considered compatible iff:*

1. ***Conditionally Compatibility**: implementations LRI_M and LRI_N are compatible.*
2. ***Specification Compatibility**: implementations LRI_M and MRI_N are compatible.*
3. ***Implementation Compatibility**: implementations MRI_M and LRI_N are compatible.*
4. ***Strong Compatibility**: implementations MRI_M and MRI_N are compatible.*

Proof (By Contradiction). To prove that analyzing compatibility of LRI_M and LRI_N is sufficient to check Conditional Compatibility of two IO MTSs, we assume that two IO MTSs are not Conditionally Compatible, and LRI_M and LRI_N are compatible. This is a contradiction as LRI_M and LRI_N are compatible implementations of M and N, which then satisfy Definition 10 of Conditional Compatibility. To prove that the compatibility of LRI_M and LRI_N is a necessary condition for two IO MTSs to be Conditionally Compatible, we assume that the two IO MTSs are Conditionally Compatible, and LRI_M and LRI_N are incompatible. In this case, according to Theorem 1, no other implementation of M can be compatible with LRI_N. Furthermore, Theorem 1 then implies that no implementation of N can be compatible with an implementation of M. This finally implies that M and N are not Conditionally Compatible, thus arriving at a contradiction.

To prove that analyzing compatibility of LRI_M and MRI_N is sufficient to check Specification Compatibility of two IO MTSs, we assume that the two IO MTSs are not Specification Compatible, and LRI_M and MRI_N are compatible. However, this is a contradiction: according to Theorem 1, if LRI_M and MRI_N are compatible then LRI_M is compatible with every implementation of N, which makes M and N Specification Compatible. To prove that the compatibility of LRI_M and MRI_N is a necessary condition for two IO MTSs to be Specification Compatible, we assume that the two IO MTSs are Specification Compatible, and LRI_M and MRI_N are incompatible. In this case, according to Theorem 2, no other implementation of M can be compatible with MRI_N, which contradicts the definition of Specification Compatibility (Definition 11). $\square$

5 Conclusions

In this paper, we revisited how compatibility should be defined for partial specifications that characterize sets of potentially valid implementations. We aimed to arrive at a foundational characterization which can be applied not only to IO MTS, but to partial-behavior models in general (including, e.g., featured transition systems [3] and disjunctive MTS [15]). To this end, we first defined four notions of partial-specification compatibility, where each notion establishes a specific relation between the specifications' implementation sets. Our definitions were specified in semantic terms, as opposed to syntactic terms, thus being more intuitive as well as more widely applicable to any model that represents a set of compliant implementations. To analyze the immediate impact of the compatibility notions, we elaborated the development processes that are allowed under the different compatibility notions, ranging from fully coupled to fully parallel development. Additionally, we introduced the concepts of the least restrictive implementation and the most restrictive implementation, which bound the space of compatible environments for an IO MTS. These concepts were then used as the foundation of low-complexity procedures for checking compatibility of two IO MTSs.

In our future work, we aim to further explore several new research avenues that are enabled by our work. In particular, we plan to research what IA-style interface refinement (as opposed to modal refinement) means in the context of IO MTS [10]. We also intend to explore whether it is possible to automatically generate an IO MTS that characterizes the subset of implementations that are compliant with another IO MTS. Finally, we aim to investigate how the IO MTS compatibility translates to development processes for systems with many components. In particular, extending pair-wise compatibility to N-way compatibility is technically simple. However the combinatorial explosion of relations between partial component specifications may require thinking of clustering them into subsystems for practical purposes. From a methodological point of view, it may be useful to link the number of clusters to the number of independent development teams, however, further research into practical ways of exploiting partial component specifications in the context of multiple development teams is required.

References

1. de Alfaro, L., Henzinger, T.A.: Interface automata. In: ESEC/FSE (2001)
2. Bauer, S.S., Mayer, P., Schroeder, A., Hennicker, R.: On weak modal compatibility, refinement, and the mio workbench. In: TACAS (2010)
3. Classen, A., Cordy, M., Schobbens, P., Heymans, P., Legay, A., Raskin, J.: Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking 39(8) (2012)
4. D'Ippolito, N., Braberman, V., Piterman, N., Uchitel, S.: The modal transition system control problem. In: FM (2012)
5. Fischbein, D., D'Ippolito, N., Brunet, G., Chechik, M., Uchitel, S.: Weak Alphabet Merging of Partial Behaviour Models. ACM TOSEM 21(2) (2012)
6. Fischbein, D., Uchitel, S.: On correct and complete strong merging of partial behaviour models. In: FSE (2008)
7. Harel, D.: Statecharts: A visual formalism for complex systems. Sci. of comp. prog. (1987)
8. Keller, R.M.: Formal verification of parallel programs. Com. of the ACM (1976)
9. Krka, I., Brun, Y., Edwards, G., Medvidovic, N.: Synthesizing Partial Component-level Behavior Models from System Specifications. In: ESEC/FSE (2009)
10. Krka, I., Medvidovic, N.: Revisiting modal interface automata. In: FORMSERA (2012)
11. Krka, I., Medvidovic, N.: Distributing refinements of a system-level partial behavior model. In: RE (2013)
12. Krka, I., Medvidovic, N.: Component-aware triggered scenarios. In: WICSA (Submitted)
13. Larsen, K.G., Nyman, U., Wasowski, A.: Modal I/O automata for interface and product line theories. In: ESOP (2007)
14. Larsen, K.G., Thomsen, B.: A Modal Process Logic. In: LICS (1988)
15. Larsen, K.G., Xinxin, L.: Equation solving using modal transition systems. In: LICS (1990)
16. Lynch, N.A., Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. PODC '87 (1987)
17. Magee, J., Kramer, J.: Concurrency: State Models & Java Programs (2006)
18. Raclet, J.B., Badouel, E., Benveniste, A., Caillaud, B., Legay, A., Passerone, R.: Modal interfaces: unifying interface automata and modal specifications. In: EMSOFT (2009)
19. Sibay, G.E., Braberman, V.A., Uchitel, S., Kramer, J.: Synthesising modal transition systems from triggered scenarios. IEEE TSE (2013)
20. Sibay, G.E., Uchitel, S., Braberman, V.A., Kramer, J.: Distribution of modal transition systems. In: FM (2012)
21. Uchitel, S., Brunet, G., Chechik, M.: Synthesis of Partial Behavior Models from Properties and Scenarios. IEEE TSE 35(3) (2009)