Component and Interface Refinement in Closed-System Specifications

Reino Kurki-Suonio

Software Systems Laboratory
Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
rks@cs.tut.fi

Abstract. A closed-system specification models a system in the context of its assumed environment. A component is then a view on the total system, where unnecessary details of other components and the external environment have been abstracted away. Starting from a crude initial model, details of components can be introduced in separate component refinements, and the resulting views can be synthesized by composition into a detailed model of the total system. In contrast to open systems, also component interfaces can be refined in this process. The original model may therefore have abstract interfaces, whose implementability with the available mechanisms is one of the concerns in component refinements. However, since component refinements may then interfere, conditions are needed for their composability. Such conditions are derived in this paper, and the application of component refinements to interface refinement is investigated.

1 Introduction

By a *closed-system* specification we understand a specification that contains also a description of the environment in which the specified component or system is intended to be used. The external environment of a reactive system can then be understood as one of the components. In contrast, an *open-system* specification describes a component or system in isolation from other components and the external environment.

"It takes two to tango," i.e., an open system does not exhibit temporal behaviors in the absence of a cooperating environment. In [7] it has therefore been argued that components of an interactive system are not proper units for structuring a specification. In particular, it seems strange to decompose a system into open components before specifying rigorously the joint behaviors that they should produce. As discussed in [9], such a decomposition does not simplify proofs, either, and may make them harder.

It should be noted, however, that the distinction between open and closed systems is to some extent in the eye of the beholder. In process algebraic approaches, for instance, an isolated process is usually understood as an open system that can be composed with other processes. From the viewpoint of this paper it is, however, a closed system, in which a generic environment cooperates to produce temporal behaviors.

Since refining an interface affects all components that communicate through it, closed-system specifications provide a natural framework for discussing interface refinement. However, working on a single large specification, which encompasses all components and the environment, need not be feasible in practice. Also, the closed-system view should not prevent working on the components separately. It should therefore be possible to treat components as partial but separately refinable and composable views, in which all unnecessary information on other components and the environment is ignored.

This leads to a specification process that is illustrated in Fig. 1. A crude model is first given at a high level of abstraction, with focus on cooperative behaviors. Partitioning into components is then imposed on this closed system, reflecting an architectural view of an eventual implementation. Having not paid attention to concrete mechanisms for component interaction in the crude model, this partitioning gives, in general, interactions that are not directly implementable with the intended communication mechanisms.

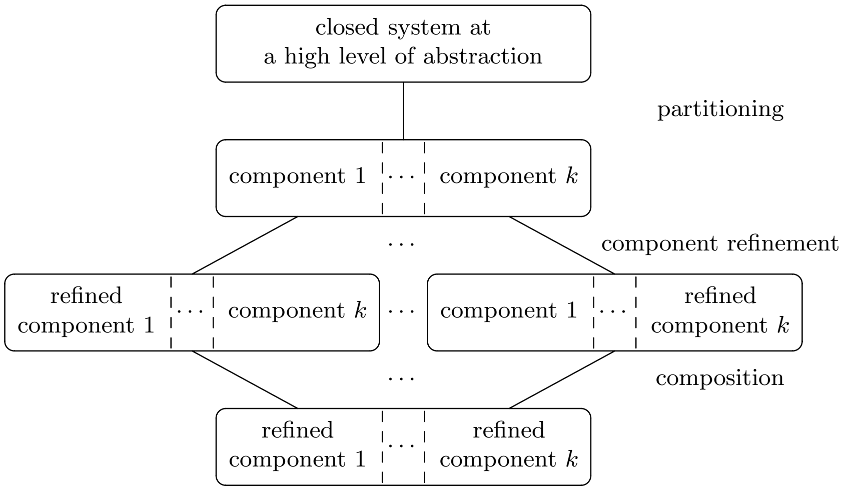

Fig. 1. Utilizing components in closed-system specification.

Next, the components in the partitioned model are refined by separate (sequences of) component refinements, which are allowed to refine also their interfaces, thereby affecting also other components in a restricted manner. Obviously, this process can be applied recursively by partitioning a component under consideration into subcomponents and refining these independently. The resulting specification is *layered* in the sense that the refinement history provides a layered structure of abstractions, where each refinement step has added a new layer to the previous ones, and composition steps have synthesized them.

The crucial problem in this process is that component refinements should remain composable into a total specification, preserving all (safety and liveness) properties introduced in them. To make the role of interface refinements in this process more concrete, consider the schematic illustration in Fig. 2, where boxes and ellipses stand for variables and actions, respectively, and A is an interface action between the two components. Although assigned to component 1, A may access and modify variables in both components, which means that some cooperation is involved in its execution. In refining component 1 one may wish, for instance, to split A into more elementary interface actions, which then affects also component 2. On the other hand, in refining component 2 one may wish, for instance, to temporarily refuse interaction A, which means disabling of an action that belongs to component 1. Obviously, the composability of such component refinements is not evident.

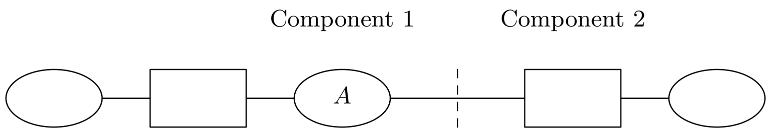

Fig. 2. Illustration of a component interface.

The formal basis for investigating these problems in this paper is Temporal Logic of Actions (TLA) [8]. In their work on TLA, Abadi and Lamport have analyzed the relationship between open and closed systems, and derived composability conditions for component refinements [1]. The problems investigated here are, however, somewhat different. Allowing component refinements to affect also external interface actions in other components makes the problem more general, but is essential for effective interface refinement. On the other hand, to achieve traceability of actions between different levels of abstraction, certain conventions are adopted for the use of TLA. Due to associated restrictions on the fairness properties that are expressible, the problem becomes manageable.

The structure of the rest of the paper is as follows. Section 2 is an introduction to TLA and its use in layered specifications. Composition of layered specifications is defined so that conventional composition of independently specified components can also be understood in terms of it. Component structure is imposed on specifications in Sect. 3. The core of the paper is in Sect. 4, where sufficient conditions are derived for the composability of component refinements, and in Sect. 5, where the approach is applied to interface refinement. The paper ends with some concluding remarks and a brief discussion on related work in Sect. 6.

2 Layered TLA-Based Specifications

The reader is assumed to be familiar with the basic notions and terminology of temporal logic. Some of the special characteristics of TLA [8] are briefly explained in this section, together with conventions that support its use in layered specifications.

2.1 TLA Formulas

TLA is a linear-time logic, where expressions are evaluated for *behaviors*, which are infinite sequences of *states. Variables* that are used to model system properties are *state functions*, which have unique values in each state. For a state function or state predicate p, its evaluation in state s will be denoted by $s[\![p]\!]$.

Corresponding to a state change, an ordered pair of states is a *step*, and *actions* are "step predicates" that are evaluated for steps. Variables in the first and the second state of a step are denoted by unprimed and primed variable names, respectively.

For an action A, state predicate *Enabled* A expresses that there exists a possible next state which, together with the given state, gives a step that satisfies A. This predicate is called the enabling condition or *guard* of action A.

A *stuttering* step, where all variables in a given set X retain their values, is denoted by $Stut_X$,

$$Stut_X \;=_{\text{def}}\; \forall x \in X : x' = x,$$

and the stuttering extension of any action A is denoted by $[A]_X$,

$$[A]_X \;=_{\text{def}}\; A \vee Stut_X.$$

In TLA, an action is allowed to appear only in *Enabled* state predicates and in contexts of the form $\Box[A]_X$. Since no "next state" operator is used, this makes the satisfaction of TLA formulas insensitive to stuttering. That is, addition and/or deletion of stuttering steps becomes inessential for behaviors, and logical implication can therefore be taken as the *refinement* relation between TLA specifications.

As usually, "$\Diamond$" will denote the dual of "$\Box$",

$$\Diamond E \;=_{\text{def}}\; \neg\Box\neg E.$$

The dual of $\Box[A]_X$ will be denoted by $\Diamond\langle A\rangle_X$, where

$$\langle A\rangle_X \;=_{\text{def}}\; A \wedge \neg Stut_X.$$

As a derived operator we will use "$\leadsto$" (leads to),

$$E_1 \leadsto E_2 \;=_{\text{def}}\; \Box(E_1 \Rightarrow \Diamond E_2),$$

and shorthand notations will be used for strong and weak *fairness* conditions with respect to actions,

$$\begin{aligned}\mathrm{SF}_X(A) \;&=_{\text{def}}\; \Box\Diamond\langle A\rangle_X \vee \Diamond\Box\neg \mathit{Enabled}\,\langle A\rangle_X,\\ \mathrm{WF}_X(A) \;&=_{\text{def}}\; \Box\Diamond\langle A\rangle_X \vee \Box\Diamond\neg \mathit{Enabled}\,\langle A\rangle_X.\end{aligned}$$

When subscripts X are understood from the context, they will be omitted in the following. Hiding of state variables by quantification will not be discussed in this paper.

2.2 Operational Expressions

An operational model can be formalized as a TLA expression of the form

$$S = P \wedge \Box[A_1 \vee \cdots \vee A_m]_X \wedge {}^{\mathrm{s}}\mathcal{F} \wedge {}^{\mathrm{w}}\mathcal{F}, \tag{1}$$

where X is the set of *variables* included in the specification, P is a satisfiable state predicate that constrains their *initial values*, A_i are *actions* that specify how the values of variables can be changed in individual steps, and ${}^{\mathrm{s}}\mathcal{F}$ and ${}^{\mathrm{w}}\mathcal{F}$ are conjunctions of *strong fairness conditions* $\mathrm{SF}_X(A_i)$ and *weak fairness conditions* $\mathrm{WF}_X(A_i)$, respectively, with respect to some of these actions. When these conditions for ${}^{\mathrm{s}}\mathcal{F}$ and ${}^{\mathrm{w}}\mathcal{F}$ are satisfied, we say that expression (1) is *operational*, or provides an *operational specification.*

By saying that a property expressed by TLA formula E holds in S we mean that $S \Rightarrow E$ is identically true. *Safety* and *liveness* properties are separated in (1) so that the first two conjuncts are pure safety properties, and the last two express pure liveness properties.

The structure of (1) allows to express S in terms of a specification language where individual actions A_i are given as syntactic units and can be referred to by their names. With individually given actions A_i we will use a script symbol $\mathcal{A}$ to stand both for the collection of actions

$$\mathcal{A} = \{A_1, \ldots, A_m\}$$

and for their logical disjunction

$$\mathcal{A} = A_1 \vee \cdots \vee A_m.$$

It will be clear from context which of the two is meant. Expression (1) can then be abbreviated to

$$S = P \wedge \Box[\mathcal{A}]_X \wedge {}^{\mathrm{s}}\mathcal{F} \wedge {}^{\mathrm{w}}\mathcal{F}. \tag{2}$$

Analogously, ${}^{\mathrm{s}}\mathcal{F}$ and ${}^{\mathrm{w}}\mathcal{F}$ will be used both for TLA formulas in (1) and for the associated sets of actions A_i for which these formulas contain fairness conditions $\mathrm{SF}(A_i)$ and $\mathrm{WF}(A_i)$, respectively.

For simplicity it will be assumed in the following that all $A_i \in \mathcal{A}$ are non-stuttering and mutually exclusive , i.e., $A_i \Rightarrow \neg \mathit{Stut}_X$, and $A_i \Rightarrow \neg A_j$ for $i \neq j$.

2.3 Refinement by Layers of Superposition

By *superposition* we understand a transformation of (2) into a formula of the same form,

$$T = Q \wedge \Box[\mathcal{B}]_Y \wedge {}^{\mathrm{s}}\mathcal{G} \wedge {}^{\mathrm{w}}\mathcal{G}, \tag{3}$$

in which

- the set of variables may be extended, $X \subseteq Y$,
- the initial condition may be strengthened, $Q \Rightarrow P$,

- each action $B_j \in \mathcal{B}$ is either a *refinement* of some $A_i \in \mathcal{A}$ called its *ancestor*,

  $$B_j \Rightarrow A_i,$$

  or a *new* action that does not modify any variables in X,

  $$B_j \Rightarrow \mathit{Stut}_X,$$

  in which case Stut_X is called its ancestor,
- ${}^{\mathrm{s}}\mathcal{G}$ and ${}^{\mathrm{w}}\mathcal{G}$ are conjunctions of strong and weak fairness conditions with respect to some actions $B_j \in \mathcal{B}$.

The situation is illustrated in Fig. 3.

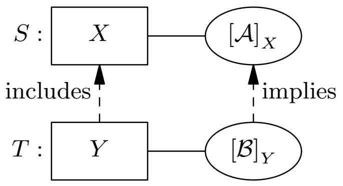

Fig. 3. Schematic illustration of superposition.

Although this definition does not say anything about liveness properties ${}^{\mathrm{s}}\mathcal{G}$ and ${}^{\mathrm{w}}\mathcal{G}$, we take it as a default that $B_j \in {}^{\mathrm{s}}\mathcal{G}$, if $A_i \in {}^{\mathrm{s}}\mathcal{F}$ for its ancestor A_i, and analogously for ${}^{\mathrm{w}}\mathcal{G}$.

From a theoretical viewpoint it is a restriction that each action in $\mathcal{B}$ is required to have a unique ancestor in $\mathcal{A} \cup \{\mathit{Stut}_X\}$, but for a specification process an explicit refinement history between actions is desirable. It is also useful to have the default convention that, in the absence of explicit refinements of A_i, action $A_i \wedge \mathit{Stut}_{Y \setminus X}$ is taken as its *default refinement*.

Provided that Q in (3) is satisfiable, T is an operational expression where all safety properties of S are preserved. In general it is not a refinement of S, and the liveness properties of S, expressed by ${}^{\mathrm{s}}\mathcal{F}$ and ${}^{\mathrm{w}}\mathcal{F}$, need not be satisfiable by any ${}^{\mathrm{s}}\mathcal{G}$ and ${}^{\mathrm{w}}\mathcal{G}$ that are permissible in (3). For default refinements it is, however, obvious that the associated liveness properties are preserved by default fairness conditions in (3).

When operational specifications are derived in incremental steps of superposition refinement, preservation of safety properties is guaranteed by construction, while proof obligations are obtained for preservation of liveness properties. At each stage we then have a correct view of the total system at some level of abstraction, where some parts of the system are given only in an abstract representation, and nondeterminism is utilized for a conservative approximation of actions. Such a specification process leads to a *layered specification*, by which we understand an operational TLA expression together with the superposition refinement history by which it has been developed.

2.4 Data Refinement in Superposition

In refinement it is often necessary to replace “abstract” data structures, which are suitable for mathematical manipulation, by “concrete” data structures that are more appropriate for efficient implementation.

Variables for new data representation can be introduced in superposition, but old variables cannot be removed. However, if one proves an invariant

$$\Box(x = f(y))$$

between an old variable x and other variables y, then x no longer needs explicit representation. Therefore, x can then be understood to have become a non-primitive state function, which henceforth provides only an abstract view on the concrete data structures by which it has been replaced.

Similarly, if an action with enabling guard g accesses but does not modify variable x, and invariant

$$\Box(g \Rightarrow x = f(y))$$

can be proved, x can be locally replaced by $f(y)$ in the action, making the action independent of x.

2.5 Composition of Layered Specifications

When two superposition refinements of the same specification (2) are given,

$$T_1 = Q_1 \wedge \Box[\mathcal{B}_1]_{Y_1} \wedge {}^{\mathrm{s}}\mathcal{G}_1 \wedge {}^{\mathrm{w}}\mathcal{G}_1,$$
$$T_2 = Q_2 \wedge \Box[\mathcal{B}_2]_{Y_2} \wedge {}^{\mathrm{s}}\mathcal{G}_2 \wedge {}^{\mathrm{w}}\mathcal{G}_2,$$

it is assumed that the new variables introduced in them are different, $Y_1 \cap Y_2 = X$. Two actions $B_1 \in \mathcal{B}_1$, $B_2 \in \mathcal{B}_2$, are then said to be *compatible*, if they have the same ancestor in $\mathcal{A} \cup \{Stut_X\}$. An action $B_1 \in \mathcal{B}_1$ with ancestor $Stut_X$ is also said to be compatible with $Stut_{Y_2}$, and similarly for $B_2 \in \mathcal{B}_2$ and $Stut_{Y_1}$.

By *composition* $T_1 \oplus T_2$ (see Fig. 4) we then understand the following superposition on both T_1 and T_2, and hence also on S:

$$T_1 \oplus T_2 = (Q_1 \wedge Q_2) \wedge \Box[\mathcal{B}]_{Y_1 \cup Y_2} \wedge {}^{\mathrm{s}}\mathcal{G} \wedge {}^{\mathrm{w}}\mathcal{G}, \tag{4}$$

where

- $\mathcal{B}$ consists of conjoined actions $B_i \wedge B_j$ for all compatible pairs $B_i \in \mathcal{B}_1 \cup \{Stut_{Y_1}\}$, $B_j \in \mathcal{B}_2 \cup \{Stut_{Y_2}\}$,
- ${}^{\mathrm{s}}\mathcal{G}$ consists of those actions in $\mathcal{B}$ for which an ancestor in either $\mathcal{B}_1$ or $\mathcal{B}_2$ belongs to ${}^{\mathrm{s}}\mathcal{G}_1$ or ${}^{\mathrm{s}}\mathcal{G}_2$, respectively, and
- ${}^{\mathrm{w}}\mathcal{G}$ consists of those actions in $\mathcal{B}$ for which an ancestor in either $\mathcal{B}_1$ or $\mathcal{B}_2$ belongs to ${}^{\mathrm{w}}\mathcal{G}_1$ or ${}^{\mathrm{w}}\mathcal{G}_2$, respectively.

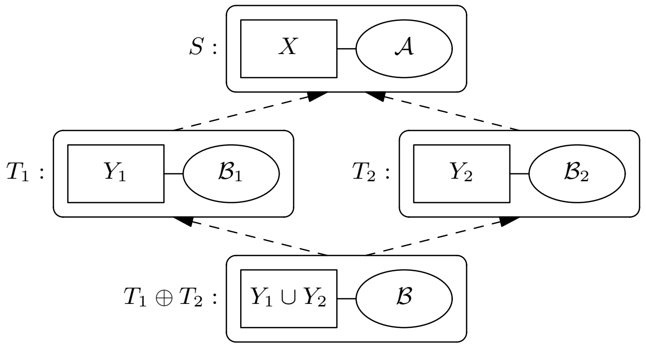

Fig. 4. Illustration of superposition relations in composition.

Obviously, "$\oplus$" is commutative and associative. Notice that it is an operation between layered specifications, not of mere TLA expressions, since the compatibility of actions in T_1 and T_2 depends on their refinement histories.

Under the above assumptions, layered specifications T_1 and T_2 are said to be *composable*, if their composition (4) is an operational expression and a refinement of both T_1 and T_2, i.e., $Q_1 \wedge Q_2$ is satisfiable, and all safety and liveness properties of both T_1 and T_2 hold in (4). In the special case of a dummy S (i.e., $X = \mathcal{A} = \emptyset$), T_1 and T_2 are independent and therefore always composable. The following Lemma is also obvious:

Lemma 1. *If T_1 and T_2 are superposition refinements of S, and no action in S is given explicit refinements in both of them, then T_1 and T_2 are composable.*

From a theoretical viewpoint one would not need to restrict to compatible pairs in the conjoined actions in (4). For a specification process it is desirable, however, that ancestor histories of actions are traceable to all levels of abstraction. For an operational interpretation it is also an advantage that conjoined actions are guaranteed not to have conflicting "assignments" to variables.

2.6 Composition of Independent Specifications

Composition of layered specifications is a generalization of composition in its conventional meaning, since the latter can be understood as the reduced case of composing mutually independent specifications. In this case it is often convenient to use parameterized actions of the form

$$A = \exists x : A(x),$$

where $A(x)$ is a step predicate that depends on parameter x.

In superposition, when $A = \exists x : A(x)$ is an ancestor of a refined action $B = \exists x : B(x)$, direct correspondence will be assumed between their parameters so that $B(x) \Rightarrow A(x)$. Obviously, if the value of x is not uniquely determined by the guard of A, further constraints can be introduced for it in B.

When “open” components are “closed” into closed-system models, parameterized actions provide effective means to describe their communication with their environments. Composition, followed by a simple superposition step, can then describe how the components act as each other’s environments.

As an example, consider independently specified components T_1 and T_2, where T_1 gives output x to its environment by action $A = \exists x : A(x)$, and T_2 receives input y from its environment by action $B = \exists y : B(y)$. Since actions A and B are compatible with each other and also with stuttering actions, composition $T_1 \oplus T_2$ then contains the conjoined action $\exists x, y : A(x) \wedge B(y)$ and also (default refinements of) both A and B. By a simple superposition step, where the default refinements are deleted[1] and the guard of the conjoined action is strengthened with $x = y$, all output from T_1 is directed to T_2 as such, and all input to T_2 is taken from T_1.

Obviously, if T_1, for instance, contains a fairness requirement with respect to action A, the above construction need not preserve this liveness property. This would be the case, for instance, if action B in T_2 would not accept all output produced by A in T_1. In the following we will study this problem in a slightly different and more general setting.

3 Components in Closed Systems

Although closed-system specifications can be used in a bottom-up manner as outlined in Sect. 2.6, their main advantages come up when the top-down direction needs to be supported. Proceeding from top down is most natural in the specification of reactive systems; in fact it sounds paradoxical to specify components of a reactive system and their interfaces before specifying what they should do together [7].

In a top-down specification process, a closed-system model is partitioned into components at some stage. In this section we discuss such partitionings and their role in the specification process.

3.1 Partitioning of State

With k components in a closed-system specification, the *global state* consists of the *local states* of these components, $s = (s_1, \ldots, s_k)$. Correspondingly, the set of variables X is partitioned into disjoint subsets

$$X = X_1 \cup \cdots \cup X_k, \quad X_i \cap X_j = \emptyset \text{ for } i \neq j,$$

so that variables in X_i are assigned to the *responsibility* of component i. Each X_i is partitioned further into *shared* and *private* variables, $X_i = X_i^{\text{shd}} \cup X_i^{\text{pvt}}$, so that external accesses (i.e., accesses by actions in other components) are allowed only to shared variables.

[1] An action is deleted in superposition by strengthening its guard to be identically false.

A crucial property of superposition is that no new write accesses can be introduced for any old variables, but read accesses can. To model situations where also external read accesses are restricted to those actions that have already been provided for that purpose, *hidden* variables X_i^{hdn} are defined as a subset of X_i^{shd} for which even no new external read accesses are allowed.

In the presence of components, the initial condition P is assumed to be *separable* into conditions on the local states of the components. Denoting the global state by a pair $s = (s_i, t)$, where the first component is the local state of component i, and t denotes the rest of the state, this assumption can be formulated as

$$(s_i, t)[\![P]\!] \wedge (u_i, v)[\![P]\!] \Rightarrow (s_i, v)[\![P]\!]. \tag{5}$$

The initial condition P is then effectively a conjunction of local conditions for the components.

3.2 Partitioning of Actions

The *responsibility* for each action is also assigned to some component, yielding a partitioning

$$\mathcal{A} = \mathcal{A}_1 \cup \cdots \cup \mathcal{A}_k, \quad \mathcal{A}_i \cap \mathcal{A}_j = \emptyset \text{ for } i \neq j.$$

This induces also an associated partitioning of fairness conditions,

$${}^{\mathrm{s}}\mathcal{F} = {}^{\mathrm{s}}\mathcal{F}_1 \cup \cdots \cup {}^{\mathrm{s}}\mathcal{F}_k, \qquad {}^{\mathrm{w}}\mathcal{F} = {}^{\mathrm{w}}\mathcal{F}_1 \cup \cdots \cup {}^{\mathrm{w}}\mathcal{F}_k,$$

so that ${}^{\mathrm{s}}\mathcal{F}_i$ and ${}^{\mathrm{w}}\mathcal{F}_i$ contain only actions in $\mathcal{A}_i$.

In principle, each TLA action involves all variables in X, independently of whether it changes their values or not. In an operational interpretation, however, an action does not need to access all variables in X. A notion of *dependence* on variables is therefore needed in the following.

Intuitively, an action A *write depends* on variable $x \in X$, if it may modify the value of x, and it *read depends* on x, if its guard $\mathit{Enabled}\, A$ or its effect on other variables may depend on the value of x. More precisely, we define these dependencies by occurrences of x' resp. x in the given textual representation (where "stuttering assignments" $x' = x$ are omitted), independently of whether these occurrences are semantically significant or not. Therefore, these dependencies fall outside of TLA, and may in some situations be changed without affecting the TLA meaning of actions.

Actions in $\mathcal{A}_i$ that are allowed to depend on shared variables in other components are called *interface actions* and are denoted by $\mathcal{A}_i^{\mathrm{ifc}}$. Other actions in $\mathcal{A}_i$ are *local* to component i. From the viewpoint of component i, actions in $\mathcal{A}_i^{\mathrm{ifc}}$ are *internal* interface actions in it, while those in $\mathcal{A}_j^{\mathrm{ifc}}$, $i \neq j$, are *external* to it.

3.3 Partitioning of Action Parameters

Parameterized actions are often useful as interface actions. In simple situations a parameter then models an input/output value that is transmitted from one component to others.

In general, parameter values need not be uniquely determined, and they may depend on variables in several components. Instead of input or output, one might then talk about "interput." Obviously, an implementation may then need complex communication, in which the components agree on an appropriate "interput" value. Partitioning into components requires, however, that the *responsibility* for each parameter is assigned to one of the components involved. Intuitively this is the component that makes the final decision on the value.

Analogously to separability of initial conditions, guards of interface actions are assumed to be *separable* with respect to parameters assigned to the responsibility of different components. More precisely, if $\exists x, y : g(x, y)$ is the guard of an interface action, where x denotes parameters that are the responsibility of one component, and y denotes the other parameters, we require

$$g(x, y) \wedge g(u, v) \Rightarrow g(x, v). \qquad (6)$$

3.4 Utilizing Components

A refinement of a partitioned specification should normally preserve or refine its component structure. Honoring this structure means that the partitioning of variables and actions into components remains compatible with their old partitionings. Similarly, compatible partitionings are required in composition of partitioned systems, and actions in different components should then be taken as mutually incompatible.

To serve a useful purpose, component structure should not be a mere add-on to a closed-system specification. In particular, partitioning into components should make it possible to work on the components independently in parallel paths of refinement. Because of interactions, component specifications are never, however, completely independent, since they always make some assumptions about their environments. Therefore, when proceeding to lower levels of abstraction, it should be possible to make also these assumptions more concrete.

To decide what the role of components should be in a (closed-system) specification process, we start from their role in implementation:

In implementation, the purpose of components is to provide modularity, where component implementations are composable into an implementation of the total system.

For specification we then adopt the analogous view:

In specification, the purpose of components is to provide modularity, where component refinements are composable into a refined specification of the total system.

Notice that this statement makes no reference to implementations. Therefore, although a specification component may focus on what corresponds to an eventual implementation component, such a correspondence is not necessary.

4 Component Refinements

In this section we discuss component refinements that serve the purpose outlined above. First we discuss simple component refinements, in which only those actions are refined that are the responsibility of the refined component, and show that these are insufficient for some practical needs. Then we formulate a robustness condition that allows also refinement of external interface actions but still guarantees composability. Throughout the section it will be tacitly assumed that component refinements conform to the given partitioning, and that new variables and actions are introduced only to the component being refined.

4.1 General Assumptions

Given an operational specification S with k components, the idea is to refine the components in S independently, yielding closed-system specifications T^i, $i = 1, \ldots, k$, which are composable into

$$T = T^1 \oplus \cdots \oplus T^k, \tag{7}$$

which then is a refinement of each T^i and hence also of S.[2]

Let P be the initial condition in S, and let Q_i be the strengthened initial condition in T^i. To guarantee separability (5) of the initial condition $Q_1 \wedge \cdots \wedge Q_k$ in T, we require

$$(s_i, t)[\![Q_i]\!] \wedge (u_i, v)[\![P]\!] \Rightarrow (s_i, v)[\![Q_i]\!]. \tag{8}$$

For interface actions we make the simplifying assumption that only one (explicit or implicit) refinement is given in T^i for each of them. (Since fairness requirements are associated with individual actions, this constrains the fairness properties that can be expressed.) No new fairness requirements are allowed for actions in other components, and all new parameters introduced in T^i must be the responsibility of component i. We also assume that only those parameters are constrained in T^i that are the responsibility of component i, and that the separability condition (6) is preserved. Therefore, if $\exists x, y : g(x, y)$ is the guard of a parameterized interface action, where x denotes the parameters that are the responsibility of component i, and $\exists x, y : h_i(x, y)$ is the corresponding refined guard in T^i, we require

$$h_i(x, y) \wedge g(u, v) \Rightarrow h_i(x, v). \tag{9}$$

4.2 Simple Component Refinements

By a *simple component refinement* of component i we understand a refinement that satisfies the above general assumptions and in which only those actions are explicitly refined that are the responsibility of component i. On account of (8) and Lemma 1 such refinements are always composable:

[2] Actions introduced as new actions in different T^i are considered mutually incompatible in this composition.

Theorem 1. *Simple component refinements are composable.*

Obviously, simple component refinements are sufficient in situations where accesses to external interface variables need not be modified. In addition, they allow restricted interface refinement by additional read dependencies on external interface variables. Still, they are insufficient in many situations that arise in practice.

As an example, consider specification of a data storage with external actions $Put(x)$ and $Get(x)$ for storing and retrieving data values x, respectively. At a high level of abstraction an abstract data structure can be used for data storage, and actions *Put* and *Get* can then be enabled whenever there is room for more data or the storage is nonempty, respectively. At a lower level of abstraction with concrete data structures, the data storage component may, however, sometimes need internal storage reorganization. In a refined specification, the need for such reorganization may therefore enable a new action *Reorg* and disable *Put* and *Get* temporarily, until reorganization by *Reorg* has taken place (see Fig. 5). Therefore, what intuitively is just a refinement of the data storage component would also refine external interface actions to it.

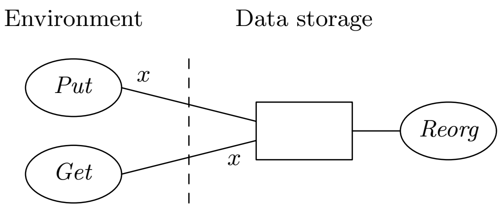

Fig. 5. Need for temporary refusal of external interface actions.

The reason for this phenomenon is that, although each interface action is assigned to some component, its execution requires cooperation from other components involved. When this cooperation is made more explicit at lower levels of abstraction, interface refinement is needed. In this example a simple kind of interface refinement is sufficient: the data storage component should be able to refuse external interface actions, which requires strengthening of their guards. We also notice that without a fairness requirement for *Reorg* such a component refinement would not be composable with environment refinements that add fairness requirements for *Put* and *Get*.

4.3 Robust Component Refinements

For reasons explained above, we allow a component refinement also to refine external interface actions to the component. Additional fairness requirements are not, however, allowed to be introduced for them.

By T_0^i we will denote the reduction of component refinement T^i where fairness assumptions are restricted to concern only local actions in component i. We then have the following Lemma:

Lemma 2. *If a property holds in the reduction T_0^i of some component refinement T^i, it also holds in the composition $T = T^1 \oplus \cdots \oplus T^k$.*

Proof. For safety properties this is obvious. Liveness properties in T_0^i are expressed by fairness requirements for local actions in component i. Since (the ancestors of) these actions cannot be refined in any other component refinement T^j, $j \neq i$, their guards are the same in T and T_0^i. Therefore, the lemma holds also for liveness properties. $\square$

As a consequence, composition T is a refinement of all T^i iff it satisfies those fairness requirements that each T^i gives for internal interface actions.

Next we formalize the idea that a component refinement can only temporarily refuse an external interface action. Let A be an external interface action to component i in specification S, let g be the guard of A, and let h_i be the guard of its refinement in component refinement T^i. (Possible parameters are existentially quantified within A, g and h_i.) We say that T^i is *insistent* on A, if condition

$$\Box\Diamond g \Rightarrow \Box\Diamond\langle A\rangle \vee \Diamond\Box(g \Rightarrow h_i) \tag{10}$$

holds in T_0^i.

A simpler condition that implies (10) and is often applicable is that h_i can be represented in the form $h_i = g \wedge r_i$ such that conditions

$$g \leadsto r_i,$$
$$\Box[r_i \Rightarrow r_i' \vee A]$$

hold in T_0^i.

A component refinement T^i is now called *robust* if, in addition to the general assumptions given above, T^i is insistent on all external interface actions. We have:

Theorem 2. *Robust component refinements are composable.*

Proof. It is sufficient to consider fairness properties of interface actions. Let $A = \exists x : A(x)$ be an internal interface action in component i in S, with x denoting its parameters collectively, and let $g = \exists x : g(x)$ and $h_i = \exists x : h_i(x)$ be the guards of A and its refinement in T^i, respectively. For each component j, $j \neq i$, for which A is an external interface action, let $h_j^{\text{ext}} = \exists x : h_j^{\text{ext}}(x)$ be the guard of its refinement in T^j.

The guard of the corresponding conjoined action in composition T is then

$$h = \exists x : (h_i(x) \wedge \bigwedge_j h_j^{\text{ext}}(x)).$$

It is now sufficient to prove that

$$\Box\Diamond h_i \Rightarrow \Box\Diamond\langle A\rangle \vee \Diamond\Box(h_i \Rightarrow h) \tag{11}$$

holds in T, since this would imply that a (weak or strong) fairness property associated with (the refinement of) A in T^i would be satisfied also in T. Since $h_i \Rightarrow g$, insistence condition (10) and Lemma 2 give us a weaker implication

$$\Box\Diamond h_i \Rightarrow \Box\Diamond\langle A\rangle \vee \Diamond\Box(h_i \Rightarrow (\exists x : h_i(x) \wedge \bigwedge_j \exists x : h_j^{\mathrm{ext}}(x))),$$

where the different parameter values for which $h_i(x)$ and $h_j^{\mathrm{ext}}(x)$ are true at the same time may be different. By parameter independence assumptions (6) and (9) there must then, however, exist also common parameter values for them, which leads to (11). $\Box$

In the example outlined in Sect. 4.2, weak fairness on action *Reorg* is obviously sufficient for making the suggested refinement of the data storage component insistent on external interface actions *Put* and *Get*. Therefore, this refinement is composable with unknown environment refinements.

5 Interface Refinement

When interactions have been defined at a high level of abstraction, they need not be realistic for direct implementation. Therefore, the need may arise to refine "abstract interactions" into more elementary "concrete interactions." In this section we discuss how component refinements can be used for this purpose.

Above it was assumed that component refinements totally conform to the original partitioning of variables and actions into components. In interface refinement this assumption needs to be relaxed. Another interesting point is that, since both internal and external interface actions can be refined in robust component refinements, the interface between two components can often be refined by refining either one of them.

By interface refinement we understand refinements that affect interface actions. The following important varieties of it can be distinguished:

- Representation of interface data can be changed by data refinement.
- External interface actions can be temporarily refused.
- The atomicity of an interaction can be refined by splitting it into more elementary actions. The responsibilities for these may be assigned to the parties involved, and the synchronization of the parties may be loosened.

In this section we will sketch out generic examples to discuss problems associated with atomicity refinement of interactions.

5.1 Changes in Responsibilities

Although each interface action is assigned to the responsibility of a specific component, its implementation requires some cooperation from each of the components involved. In refining an interface action the roles of other components may therefore become explicit in actions that are assigned to their responsibility, and this may also affect the partitioning of variables and actions. Since components have no significance in terms of TLA, composable refinements remain composable independently of such changes.

Obviously, for components to have any significance, arbitrary changes to the partitioning should not be allowed. The principles that we adopt are that a component refinement can reallocate only responsibilities of the refined component to other components, and that all such changes honor the general requirements for partitionings, even when composed with unknown component refinements. For instance, the responsibility for an action in $\mathcal{A}_i$ cannot be moved to another component if it depends on variables in X_i^{pvt}. Similarly, a necessary condition for changing a public variable to become private is that it belongs to X_i^{hdn}.

As an extreme example of changes in responsibilities, consider the following. If it is decided in an environment refinement that a real environment will be replaced by a simulated one, then all variables and actions of the environment part are moved to the system part. Such an environment refinement can then be composed with a system refinement, yielding a refined specification with a simulated environment.

Conversely, we can think of a system refinement that moves all its responsibilities to the external environment. This would reflect the decision that a separate system part will not be used, and everything will be implemented by rules and activities imposed on the environment. This demonstrates that, although an interface refinement can be carried out as a refinement of one component, its feasibility cannot be judged without considering all parties concerned.

5.2 Example: Simplifying an Interface Action

In a high-level specification an external interface action may directly execute some computation that an implementation should assign to local actions. As an example, consider a closed system consisting of an environment component (component 1) and a system component (component 2), where system variable $x \in X_2^{\mathrm{hdn}}$ (initialized as 0) is used to accumulate information given by an environment action $A \in \mathcal{A}_1^{\mathrm{ifc}}$,

$$\begin{aligned} &A(i : \text{integer}) : \mathit{true} \\ &\qquad\qquad \to x' = f(i, x), \end{aligned}$$

where f denotes some integer-valued function. In the notations, an arrow is used to separate the guard of an action from its "assignments," and "stuttering assignments" are not given explicitly.

The problem with this interface is that all computing, i.e., each evaluation of function f, is done by the environment, which therefore also needs access to the accumulated value x. A robust system refinement can, however, be given, which refines the interface so that variable x is effectively removed, and evaluation of f is moved to take place in a system action (see Fig. 6).

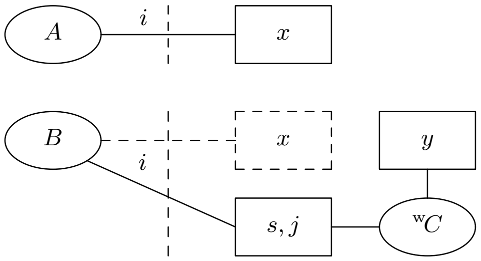

Fig. 6. Example of interface simplification.

As for variables, new system variables $s, j \in X_2^{\mathrm{shd}}$ and $y \in X_2^{\mathrm{pvt}}$ are introduced in this refinement, initialized as *true*, 0 and 0, respectively. Interface action A, which is external to the system component, is refined into B,

$$\begin{aligned} B(i : \text{integer}) : & \; A(i) \\ & \wedge s = \mathit{true} \\ & \rightarrow j' = i \\ & \wedge s' = \mathit{false}, \end{aligned}$$

and a new local system action ${}^{\mathrm{w}}C$ is introduced,

$$\begin{aligned} {}^{\mathrm{w}}C : & \; s = \mathit{false} \\ & \rightarrow y' = f(j, y) \\ & \wedge s' = \mathit{true}. \end{aligned}$$

Prefix ${}^{\mathrm{w}}$ on ${}^{\mathrm{w}}C$ expresses a weak fairness requirement, which ensures that this refinement is, indeed, a robust component refinement of the system part.

Obviously, invariant $\Box(x = (\text{if } s = \mathit{true} \text{ then } y \text{ else } f(j, y)))$ now makes x redundant and, since no additional dependencies on x can be introduced in potential environment refinements, x can be removed. Evaluation of function f has then been effectively moved from the environment to the system part, and environment access to x has been removed.

5.3 Example: Refinement of Communication

The normal method to refine the atomicity of an action A is to introduce new actions which together with a refined action B accomplish what was originally done by A alone. Compared to A, the enabling of B is then delayed until the new actions have done all the preparatory work. Here we illustrate how this technique can be used for refining the atomicity of communication between components.

Consider the situation illustrated in the upper part of Fig. 7. There, the environment part gives an integer x to the system part in an atomic action A. For simplicity it is assumed that action A updates no environment variables. Removing this restriction will be discussed below.

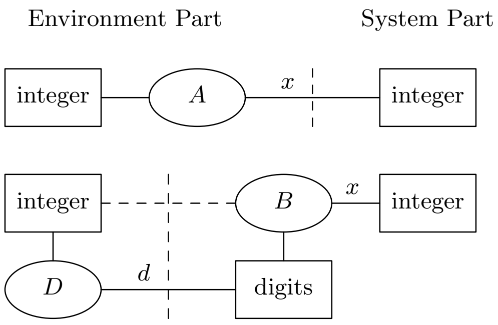

Fig. 7. Example of atomicity refinement.

Assuming that an implementation cannot transmit an entire integer atomically, a refinement is needed where the digits of x are transmitted one by one. This can be expressed as an environment refinement, which can be outlined as follows (see lower part of Fig. 7). The digits d of x are given to the system part by a new environment action D. Once all of them have been transmitted, action B, which is a refinement of A, can reconstruct x from them. By proving the invariant that this integer is indeed x, the dependence of B on environment variables (shown by a dashed line) can be removed, and B can be changed into a local system action.

5.4 Loosening of Component Synchronization

In the previous example it was assumed that action A did not update any environment variables. Otherwise the dependence of B on environment variables could not have been removed. In general, an interface action may update variables in all parties involved, and an implementation may therefore need communication in each direction, and refinement of atomicity then needs loosening of synchronization between the components.

To sketch how this affects the refinement of interface atomicity, consider a situation where an interface action A models two-way communication between two components, updating variables x and y in them (see upper part in Fig. 8).

To get rid of synchronous updating of x and y, one of them (x) has to be transformed into a non-primitive state function. Therefore, let z be a new variable whose value will "almost always" agree with that of x. The functions of action A can then be split into more elementary actions as follows (see lower

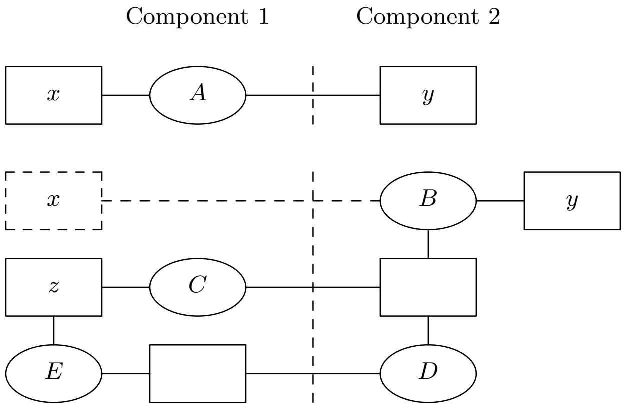

Fig. 8. Loosening of synchronization in two-way communication.

part of Fig. 8): a new action C transmits the required values to component 2, action B, which is a refinement of A, then updates y accordingly and computes the feedback, which is subsequently returned to component 1 by another new action D, and still another new action E finally updates z.

For the correctness of such a refinement it is essential that x has, indeed, become a non-primitive state function that needs no explicit representation, i.e., x is needed in component 1 only when its value is present in z. Another aspect that needs attention is that the enabling of A (and hence also of B) may depend on variables in component 2. Therefore, action C may be executed also in situations where its effects need to be canceled.

6 Concluding Remarks

The idea of layered specifications, as presented in Sect. 2, was developed in connection with the DisCo specification language [4, 5]. Technically, an import clause in DisCo takes the composition of one or more layered specifications as a basis for superposition. Partitioning into components and component refinements, as discussed in this paper, are not supported by the language. The layered structure of DisCo specifications has been utilized in an aspect-oriented manner, where different aspects of the specification are addressed in parallel paths of superposition refinement. Preservation of liveness properties in the composition of the resulting refinements has been treated on a case-by-case basis.

Another approach to TLA-based specifications has been developed in cTLA [3]. Instead of layered specifications, cTLA supports composition of independent specifications (and subsequent superposition) in the manner outlined in Sect. 2.6. Similarly to DisCo, preservation of liveness properties needs to be considered on a case-by-case basis. In the event-based formal framework of LOTOS, where fairness properties are not specified, essentially the same technique is known as the constraint-oriented specification style [2].

Object-oriented concepts have not been used in this paper. In DisCo, state variables are given in terms of classes and objects, actions are parameterized by the objects that can "participate" in them, and inheritance of single-object methods has been generalized into inheritance of capabilities to participate in such multi-object actions [6]. DisCo specifications are therefore patterns of interactive object systems. The results of this paper can be generalized also to this object-oriented situation, where components are collections (or patterns) of objects with associated actions.

Unlike conventional approaches to modularity, layered specifications make it possible to start formal modeling at a high level of abstraction, where component interfaces have not yet been fixed. Components can then be considered as different "aspects," on which the focus is set in parallel refinement paths. In particular, component interfaces can also be refined in this process. Conditions for the composability of such component refinements have been investigated in this paper. The approach seems natural for codesign, for instance, where the joint activities of the components should be specified before deciding on exact partitioning and concrete interfaces [10].

The viewpoint of this paper has been that of a top-down specification process. Therefore, attention has not been paid to techniques for component reuse. Layered specifications make it possible, however, to reuse specifications also at a high level of abstraction, where the operands of composition do not focus on implementation components. Need for reuse at such levels is apparent in design patterns, for instance, to which layered specifications and the object-oriented inheritance mechanism of DisCo have been applied in [11].

References

[1] Abadi, M., Lamport, L.: Conjoining specifications. ACM TOPLAS **17** (May 1995) 507–534

[2] Bolognesi, T., Brinksma, E.: Introduction to the ISO specification language LOTOS. Computer Networks and ISDN Systems **14** (1987) 25–59

[3] Herrmann, P., Krumm, H.: Compositional specification and verification of high-speed transfer protocols. In Protocol Specification, Testing and Verification XIV (Eds. S. T. Vuong and S. T. Chanson), Chapman & Hall 1994, 339–346

[4] Järvinen, H.-M.: The Design of a Specification Language for Reactive Systems. Tampere University of Technology, Publication 95, 1992

[5] Järvinen, H.-M., Kurki-Suonio, R., Sakkinen, M, Systä, K.: Object-oriented specification of reactive systems. Proc. 12th Int. Conf. on Software Eng., IEEE Computer Society 1990, 63–71

[6] Kurki-Suonio, R.: Fundamentals of object-oriented specification and modeling of collective behaviors. In Object-Oriented Behavioral Specifications (Eds. H. Kilov and W. Harvey), Kluwer 1996, 101–120

[7] Kurki-Suonio, R., Mikkonen, T.: Harnessing the power of interaction. In Information Modelling and Knowledge Bases X (Eds. H. Jaakkola, H. Kangassalo and E. Kawaguchi), IOS Press 1999, 1–11

[8] Lamport, L.: The temporal logic of actions. ACM TOPLAS **16** (May 1994) 872–923

[9] Lamport, L.: Composition: a way to make proofs harder. Compaq Systems Research Center, Technical Note 1997-030a, January 1998
[10] Mikkonen, T.: A development cycle for dependable reactive systems. In Proc. IFIP International Workshop on Dependable Computing and its Applications, DCIA98. Available at `http://www.cs.wits.ac.za/research/workshop/ifip98.html`
[11] Mikkonen, T.: Formalizing design patterns. Proc. 20th Int. Conf. on Software Eng., IEEE Computer Society 1998, 115–124