From Action System to Distributed Systems: The Refinement Approach
Contents

List of Figures ix
List of Tables xi

I This is What a Part Would Look Like 1

1 A Contract-Based Approach to Ensuring Component Interoperability in Event-B 3
   Linas Laibinis and Elena Troubitsyna
   1.1 Introduction .............................................. 3
   1.2 Background: Event-B ................................. 5
      1.2.1 Modelling and Refinement in Event-B .......... 5
      1.2.2 Modelling Modular Systems in Event-B .......... 7
   1.3 From Event-B Modelling to Contracts .................. 11
      1.3.1 Contracts ........................................ 11
      1.3.2 From a Module Interface to a Component Contract .. 12
   1.4 Example: an Auction System ........................... 13
      1.4.1 Initial Model ................................... 14
   1.5 Conclusions ............................................. 19

Bibliography 21
List of Figures

1.1 Event-B machine and context components ............... 5
1.2 Before-after predicates .................................. 6
1.3 Module interface ......................................... 8
1.4 Component contract ...................................... 11
1.5 Interface Component ...................................... 17
1.6 The Seller class contract ................................. 18
List of Tables
Part I

This is What a Part Would Look Like
1.1 Introduction

Ensuring component interoperability constitutes one of the main challenges in the component-based development approach [10]. The approach relies on a composition of reusable software components (or services) to implement a desired functionality [19]. While composing them, we should ensure that the components are compatible not only at the interface but also at the semantic level. Essentially, it requires for the components to share the knowledge about their globally observable behaviour and properties. The most popular approach to representing such knowledge is by defining component contracts.

Usually a contract is expressed as an assumption-guarantee pair [11]. The assumption part postulates the properties that the component’s environment should satisfy, while the guarantee part defines the properties that must be satisfied by the component itself.
The design by contract approach proposed by Meyer [16] facilitates structuring of software into encapsulated modules with contracts regulating component interactions. While the benefits of the contract-based approach are evident, defining the contracts themselves is often a challenging task, especially in the development of decentralised systems with complex component interdependencies. In this paper, we propose a refinement-based approach facilitating definition of contracts and ensuring component interoperability.

The formal basis of the proposed approach is within Event-B [1] and its modularisation extension [13]. Event-B is a formal approach for designing correct-by-construction distributed systems. The main development technique of Event-B – refinement – allows the designers to transform an abstract specification into a detailed model through a chain of correctness-preserving model transformations. Each refinement step is verified by proofs guaranteeing that the refined model preserves the externally observable behaviour and does not introduce new deadlocks. Modelling and verification in Event-B is automated by an industrial-strength tool – the Rodin platform [21].

Refinement allows us to formally define relations between models representing the system behaviour at different levels of abstraction. Hence it constitutes a suitable mechanism for establishing relationships between the system-level properties and the behaviour of system components.

The main idea behind our approach is to derive component contracts from a formal Event-B specification of the overall system. We start from an abstract centralised specification of the system. The initial model relies on the shared global state and abstracts away the communication between the components. The interoperability properties are fairly transparent at this level and easy to define. In the refinement process, while elaborating on the system behaviour and properties, we introduce a representation of inter-component communication, distribute the global state space between the components and decouple them. Finally, we decompose the obtained system specification into independent modules. While deriving the module interfaces, we at the same time define their contracts. Since decomposition is a special kind of refinement step, we guarantee that the components remain interoperable under the derived contracts.

We believe that the proposed approach facilitates the rigorous development of complex component-based systems and enhances confidence in the correctness of the overall system design. It allows us to propagate the system level properties into the component contracts and ensure component interoperability.

The paper is structured as follows: in Section 1.2, we present our main modelling framework – Event-B and its modularisation extension. In Section 1.3, we demonstrate how to derive contracts from Event-B models using the modularisation extension. In Section 1.4, we give a small illustrative example – an auction system. Finally, in Section 1.5, we overview the proposed approach and discuss the related work.
1.2 Background: Event-B

In this section, we overview our modelling framework – Event-B. The Event-B formalism [1] is a state-based formal approach that promotes the correct-by-construction development paradigm and formal verification by theorem proving. Event-B is a specialisation of the B Method that facilitates modelling of event-based (reactive) systems by incorporating the ideas of the Action Systems formalism [3] into the B Method.

1.2.1 Modelling and Refinement in Event-B

In Event-B, a system specification (model) is defined using the notion of an abstract state machine [1]. An abstract state machine encapsulates the model state, represented as a collection of model variables, and defines operations on this state. Therefore, it describes the dynamic part (behaviour) of the modelled system. Usually a machine also has the accompanying component, called context, which contains the static part of the model. In particular, a context can include user-defined carrier sets, constants and their properties, which are given as a list of model axioms. A general form of Event-B models is given in Figure 1.1.

The machine is uniquely identified by its name $M$. The state variables, $v$, are declared in the Variables clause and initialised in the Init event. The variables are strongly typed by the constraining predicates $I$ given in the Invariants clause. The invariant clause might also contain other predicates defining properties that should be preserved during system execution.

The dynamic behaviour of the system is defined by the set of atomic events specified in the Events clause. Generally, an event can be defined as follows:

\[
\text{any } lv \text{ where } g \text{ then } S \text{ end,}
\]

where $lv$ is a list of new local variables (parameters), the guard $g$ is a state...
From Action System to Distributed Systems: The Refinement Approach

<table>
<thead>
<tr>
<th>Statement (S)</th>
<th>$BA_S(s, c, lv, x, y, x')$</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>$x = x \land y' = y$</td>
</tr>
<tr>
<td>$x := E(s, c, lv, x, y)$</td>
<td>$x' = E(s, c, lv, x, y) \land y' = y$</td>
</tr>
<tr>
<td>$x \in Set$</td>
<td>$x' \in Set \land y' = y$</td>
</tr>
<tr>
<td>$x : P(s, c, lv, x, y, x')$</td>
<td>$P(s, c, lv, x, x') \land y' = y$</td>
</tr>
<tr>
<td>$S_1 \parallel S_2$</td>
<td>$BA_{S_1} \land BA_{S_2}$</td>
</tr>
</tbody>
</table>

FIGURE 1.2: Before-after predicates

predicate, and the action $S$ is a statement (assignment). In the case when $lv$ is empty, the event syntax becomes \textbf{when $g$ then $S$ end}. If $g$ is always true, the syntax can be further simplified to \textbf{begin $S$ end}.

The occurrence of events represents the observable behaviour of the system. The guard defines the conditions under which the action can be executed, i.e., when the event is enabled. If several events are enabled at the same time, any of them can be chosen for execution nondeterministically. If none of the events is enabled then the system deadlocks.

In general, the action of an event is a parallel composition of statements (assignments). The statements can be either deterministic or nondeterministic. A deterministic assignment, $x := E(x, y)$, has the standard syntax and meaning. A nondeterministic assignment is denoted either as $x \in Set$, where $Set$ is a set of values, or $x : P(x, y, x')$, where $P$ is a predicate relating initial values of $x$ and $y$ to some final value of $x'$. As a result of such a assignment, $x$ can get any value belonging to $Set$ or according to $P$.

Semantics of an Abstract Model. The semantics of Event-B actions is defined using before-after (BA) predicates [1]. A before-after predicate describes a relationship between the system states before and after an execution of an event action. The definitions of a BA for different action statements are shown in Figure 1.2. Here $x$ and $y$ are disjoint lists (partitions) of state variables, and $x'$, $y'$ represent their values in the state after the action execution. Moreover, $lv$ refers to the local event variables, where $s$ and $c$ stand for respectively the sets and constants defined in the model context.

The notion of a BA predicate can be easily generalised to formally define model events. For an event $e$ of the form \textbf{any $lv$ where $g$ then $S$ end}, its BA predicate is as follows:

$$BA_e(s, c, x, y, x') = \exists lv. g(s, c, lv, x, y) \land BA_S(s, c, lv, x, y, x')$$

The semantics of a whole Event-B model is formulated as a number of \textit{proof obligations}, expressed in the form of logical sequents. Below we present only the most important proof obligations that should be verified for the initial and refined models. The full list of proof obligations can be found in [1].

An initial Event-B model should satisfy the event feasibility and invariant preservation properties. For each event of the model – $e_i$ – its feasibility means
that, whenever the event $e_i$ is enabled, its before-after predicate (BA) is well-defined, i.e., there exists some reachable after-state:

$$A(s, c), \ I(s, c, v), \ ge_i(s, c, lv, v) \vdash \exists v' \cdot BA_{e_i}(s, c, lv, v, v')$$

where $A$ are the model axioms, $I$ is the model invariant, $ge_i$ is the event guard, $d$ are the model sets, $c$ are the model constants, $lv$ are the local event variables, and $v, v'$ are the variable values before and after event execution.

Each event $e_i$ of an initial Event-B model should also preserve the given model invariant:

$$A(s, c), \ I(s, c, v), \ ge_i(s, c, lv, v), \ BA_{e_i}(s, c, lv, v) \vdash I(s, c, v')$$

Since the initialisation event $Init$ has no initial state, local variables, and guard, its proof obligation is simpler:

$$A(s, c), \ BA_{Init}(s, c, v') \vdash I(s, c, v')$$

**Semantics of a Refined Model.** Event-B employs a top-down refinement-based approach to system development. Development starts from an abstract system specification that models the most essential functional requirements. While capturing more detailed requirements, each refinement step typically introduces new events and variables into the abstract model. These new events correspond to stuttering steps that are not visible at the abstract level.

To verify the correctness of a refinement step, we need to prove a number of proof obligations for the refined model. Intuitively, those proof obligations allow us to demonstrate that the refined machine does not introduce new observable behaviour, more specifically, that concrete states are linked to the abstract ones via the given (gluing) invariant of the refined model. All proved properties of the abstract model are automatically "inherited" by the refined one. For brevity, we omit here discussion of these proof obligations. They can be found in [1].

The Event-B refinement process allows us to gradually introduce implementation details, while preserving functional correctness during stepwise model transformation. The model verification effort, in particular, automatic generation and proving of the required proof obligations, is significantly facilitated by the provided tool support – the Rodin platform [21].

### 1.2.2 Modelling Modular Systems in Event-B

Recently the Event-B language and tool support have been extended with a possibility to define modules [13, 17] – components containing groups of callable operations. Modules can have their own (external and internal) state and the invariant properties. The important characteristic of modules is that they can be developed separately and then composed with the main system.

**Module Structure.** A module description consists of two parts – module
interface and module body. Let $M$ be a module. The module interface is a separate Event-B component. It allows the user of the module $M$ to invoke its operations and observe the external variables of $M$ without having to inspect the module implementation details. The module interface consists of the module interface description $MI$ and its context $MI_{Context}$. The context defines the required constants $c$ and sets $s$. The interface description consists, respectively, of the external module variables $w$, the external module invariant $MI_{Inv}(s, c, w)$, and a collection of module operations, characterised by their pre- and postconditions, as shown in Figure 1.3. The primed variables in the operation postcondition stand for the variable values after operation execution, while the predefined variable $res$ refers to the operation result to be returned.

In addition, a module interface description may contain a group of standard Event B events under the \textsc{Process} clause. These events model the autonomous module thread of control, expressed in terms of their effect on the external module variables. In other words, the module process describes how the module external variables may change between operation calls.

A module development always starts with the design of an interface. After an interface is formulated, it cannot be altered in any manner. This ensures correct relationships between a module interface and its body, i.e., that the specification of an operation call is recomposable with an operation implementation. A module body is an Event-B machine. It implements each operation described in the module interface by a separate group of events. Additional proof obligations are generated to verify the correctness of a module. They guarantee that each event group faithfully satisfies the given pre- and post-conditions of the corresponding interface operation.

\textbf{Importing of a Module.} When the module $M$ is imported into another Event-B machine, this is specified by a special clause \textsc{Uses} in the importing machine, $N$. As a result, the machine $N$ can invoke the operations of $M$ as well as read the external variables of $M$ listed in the interface $MI$.

To make a module interface generic, in $MI_{Context}$ we can define some
abstract constants and sets (types). Moreover, the interface MI itself may be parameterised with the constant \( id \), which is used as an unique identifier for a module instance within the interface. All such data structures become module parameters that can be instantiated when a module is imported. The concrete values or constraints needed for module instantiation are supplied within the USES clause of the importing machine. Alternatively, the module interface can be extended with new sets, constants, and the properties that define new data structures and/or constrain the old ones. Such an extension produces a new, more concrete module interface. Via different instantiation of generic parameters the designers can easily accommodate the required variations when developing components with similar functionality. Hence module instantiation provides us with a powerful mechanism for reuse.

We can create several instances of a given module and import them into the same machine. Different instances of a module operate on disjoint state spaces. Identifier prefixes can be supplied in the USES clause to distinguish the variables and operations of different module instances or those of the importing machine and the imported module. Alternatively, the pre-defined set can be supplied as an additional parameter. In the latter case, module instances are created for each element of the given set. The syntax of USES then becomes as follows:

\[
\text{USES } < \text{module interface}> \text{ as } < \text{prefix}> \\
\text{or} \\
\text{USES } < \text{module interface} > [ < \text{constant set} > ]
\]

**Semantics of a Module Interface.** Similarly to a machine component, the semantics of an interface component is defined by a number of proof obligations. The module initialisation must establish the module invariant \( M_{\text{Inv}} \): 

\[
M_{\text{Init}}(s,c,mv') \vdash M_{\text{Inv}}(s,c,mv') \quad \text{(MOD_INIT)}
\]

Let us assume \( \text{Oper}_i, \ i \in 1..N \), is one of module operations. The module invariant \( M_{\text{Inv}} \) should be preserved by each operation execution:

\[
M_{\text{Inv}}(s,c,mv), \ \text{Pre}_i(s,c,p,mv), \ \text{Post}_i(s,c,p,mv,\text{mv}',\text{res}) \vdash M_{\text{Inv}}(s,c,\text{mv}') \quad \text{(MOD_INV1)}
\]

where \( \text{Pre}_i \) and \( \text{Post}_i \) are respectively the precondition and postcondition of \( \text{Oper}_i \).

Let us assume \( \text{Ev}_j, \ j \in 1..K \), is one of module process events. The module invariant \( M_{\text{Inv}} \) should be also preserved by each such event:

\[
M_{\text{Inv}}(s,c,mv), \ \text{BA}_j(s,c,lv,mv,\text{mv}') \vdash M_{\text{Inv}}(s,c,\text{mv}') \quad \text{(MOD_INV2)}
\]

where \( \text{BA}_j \) is the before-after predicate of \( \text{Ev}_j \).

Finally, there is a couple of feasibility proof obligations for each \( \text{Oper}_i, \ i \in \text{1..N} \), and each \( \text{Ev}_j, \ j \in \text{1..K} \):
1. Firstly, the operation precondition should be true for at least some of parameter values:

\[ M_{\text{Inv}}(s, c, mv) \vdash \exists p. Pre_i(s, c, p, mv) \]  

(MOD\_PARS)

Secondly, at least some operation post-state containing the required result must be reachable:

\[ M_{\text{Inv}}(mv), Pre_i(p, mv) \vdash \exists (mv', res). Post_i(p, mv, mv', res) \]  

(MOD\_RES)

**Semantics of an Operation Call.** A machine importing a module instance operates on the extended state consisting of its own variables \(v\) and the module variables \(mv\). The module state can be updated in event actions only via operations calls. The semantics of an event containing an operation call is as follows.

Let us consider the model event \(E_c\) that contains a call to the module operation \(Op\) with the given arguments \(args\), i.e., it is of the form

\[
\text{any } lv \text{ where } g \text{ then } S[Op(args)] \text{ end.}
\]

The BA predicate of such an event can be defined as follows:

\[
BA_{E_c}(s, c, v, mv, v', mv') = \exists (lv, res, new_mv). g(s, c, lv, v, mv) \land \\
Post(s_{MI}, c_{MI}, args, mv, new_mv, res) \land \\
BA_{S*}(s, c, lv, v, mv, res, v') \land (mv' = new_mv),
\]

where \(S^*\) is \(S\) with all the occurrences of \(Op(args)\) replaced by \(res\), while \(s_{MI}\) and \(c_{MI}\) are respectively the sets and constants defined in the module interface context. Once this is done, we can rely on the existing proof semantics to verify the invariant preservation, event simulation and other required properties.

Moreover, we need an additional proof obligation to ensure call correctness by checking that the operation precondition holds at the place of an operation call:

\[ g(s, c, lv, v, mv), \text{Inv}(s, c, v, mv), M_{\text{Inv}}(s_{MI}, c_{MI}, mv) \vdash Pre(s_{MI}, c_{MI}, args, mv) \]  

(CALL\_CORR)

The modularisation extension of Event-B facilitates formal development of complex systems by allowing the designers to decompose large specifications into separate components and verify system-level properties at the architectural level. Next we demonstrate how to define contracts based on the modularisation extension of Event-B.
A Contract-Based Approach to Ensuring Component Interoperability in Event-B

1.3 From Event-B Modelling to Contracts

1.3.1 Contracts

Contracts constitute a widely-used approach to ensuring component interoperability [10, 16]. Via their contracts, the components share the knowledge about their globally observable behaviour and properties. In the component-based frameworks, a contract is usually defined as an assume-guarantee pair [11]. The assumption part defines the properties that the environment must satisfy, while the guarantee part expresses the properties that should be satisfied by the component itself.

The development methodology based on refinement allows us to express system-wide properties, which can significantly facilitate definition of component contracts. These properties are defined as model invariants. Our goal is to find a mechanism for propagating the relevant system-level properties into the component contracts.

To ensure component interoperability, in our definition of a contract, we will describe the component interface, interactions between the component and its environment, as well as abstractions for the expected autonomous behaviour of a component. The generic form of a contract for a class of components $C$ is presented in Fig.1.4.

Since one of the strengths of the component-based development is the support for component reuse, a single contract can represent a family (class) of components that might differ by their implementations, internal behaviour etc. Hence, in our definition of a component contract given in Fig.1.4, we explicitly state that the contract is defined for a class of components.

The EXTERNAL VARIABLES clause defines the globally observable state of components (represented by a collection of variables $v$). The IN-
VARIANT clause defines the types of the external variables as well as the properties always maintained over them. The initial state of a component is constructed according to the state predicate defined in the INITIALISATION clause. The ACTIONS part of the contract regulates the dynamic behaviour of the component. It is defined as a pre- and post-condition pair for each operation of the component that changes its externally observable state. The operations might have parameters that are defined in the params clause. Finally, the class itself may be parameterised with the constant id to be used as an unique identifier for class instances.

It is easy to note, that the definition of a contract closely resembles the definition of a module interface given in Fig. 1.3. Indeed, a module interface defines the globally observable state of the component, its properties in the INVARIANT clause and callable operations as pre- and post-condition pairs. The only difference is in representing the internal (autonomous) component behaviour defined in the PROCESS clause. Next we will address this issue and establish a correspondence between the definitions of a module interface and a contract.

1.3.2 From a Module Interface to a Component Contract

In our definition of the module, the PROCESS clause defines the autonomous internal behaviour of the component, i.e., specifies how the external component state may change between operation calls. The behaviour is modelled by a set of events. To transform this representation into the prepostcondition format, let us give an alternative definition of an Event-B event.

Essentially, an event $e$ of the form any $a$ where $G_e$ then $BA_e$ end is a relation describing the corresponding state transformation from $\sigma$ to $\sigma'$, such that

$$e(\sigma, \sigma') = \exists \rho \cdot I(s,c,\sigma) \land G_e(s,c,\rho,\sigma) \land BA_e(s,c,\rho,\sigma,\sigma'),$$

where $\rho$ represents the local event state. Here we treat the model invariant $I$ as an implicit event guard. Note that, due to the possible presence of non-determinism, the successor state $\sigma'$ is not necessarily unique.

In other words, the semantics of a single model event is given as a binary relation between pre- and post-states of the event. To represent this relationship, we define two functions before and after in a way similar to [12, 20]:

$$\text{before}(e) = \{(\rho, \sigma) \mid I(s,c,\sigma) \land G_e(s,c,\rho,\sigma)\}$$

and

$$\text{after}(e) = \{\sigma' \mid \exists \rho, \sigma \cdot I(s,c,\sigma) \land G_e(s,c,\rho,\sigma) \land BA_e(s,c,\rho,\sigma,\sigma')\}.$$ 

One can see that, for a given event $e$ and any state $\sigma \in \Sigma$, $e$ is enabled in $\sigma$ if and only if $(\rho, \sigma) \in \text{before}(e)$ for some possible value of the local variables.
A Contract-Based Approach to Ensuring Component Interoperability in Event-B

\( \rho \). Essentially, the functions \texttt{before} and \texttt{after} define the domain and range of the underlying semantic event definition as a before-after relation between the states.\(^1\)

The alternative representation of an event allows us to express both externally callable operations and autonomic component behaviour in the pre- and postcondition form. To be more precise, each callable interface operation

\[ \text{Op}_i = \text{any } p_i \text{ pre } \text{Pre}_i(...) \text{ post } \text{Post}_i(...) \text{ end} \]

is directly translated into the action

\[ \text{A}_i = \text{params } p_i \text{ pre } \text{Pre}_i \text{ post } \text{Post}_i, \]

while each event in the \texttt{PROCESS} clause

\[ \text{Ev}_j = \text{any } lv_j \text{ where } G_j(...) \text{ then } \text{BA}_j(...) \text{ end} \]

is mapped into the following action:

\[ \text{A}_j = \text{params } lv_j \text{ pre before(Ev}_j \text{) post after(Ev}_j \text{),} \]

where \texttt{before} and \texttt{after} are the functions defined above.

We have established the correspondence between the definitions of a module interface and a contract. We believe that the modularisation extension of Event-B has provided us with a suitable basis for deriving component contracts. Indeed, it allows us to easily derive the definitions of all parts of a contract from the corresponding definition of a module interface. Since the component contracts are derived from a specification of the overall system, our approach supports "interoperability by construction". It ensures that the components composed to achieve the specified system functionality are interoperable provided they comply with the derived contracts. In the next section, we illustrate the proposed approach by an example – an auction system.

1.4 Example: an Auction System

In this section, we illustrate the proposed approach by an example – a simple electronic auction. We start from a centralised abstract system model of an auction, then introduce an abstract model of the communication mechanism for sending and receiving different types of requests, and, finally, decompose the model into specifications of the interface interfaces (contracts) of the involved components.

\(^1\)We slightly modified the \texttt{before} definition comparing to its original one, where

\[ \text{before}(e) = \{ \sigma | \exists \rho : I(s, c, \sigma) \land G_e(s, c, \rho, \sigma) \} \],

to make it more suitable for defining component contracts.
14 From Action System to Distributed Systems: The Refinement Approach

1.4.1 Initial Model

The initial auction specification describes the activities of components of three different types: a seller, a buyer, and a manager. There could be any number of sellers and buyers participating in the system. However, there should be only one manager, which keeps the information about the current auction state and controls validity of the auction operations.

The auction specification describes the following allowed scenario for a seller, a buyer, and a manager. The scenario is initiated by a seller, who announces an item to be sold at the auction. Once the item announcement is received by the manager, the bidding process for this particular item starts.

Any active buyer can make its bid. However, only higher bids (for a particular item) are accepted by the manager. After a predefined number of bids, the bidding process is stopped and the winner (i.e., a bidder with the highest bid) is decided by the manager. Once the payment from the winner is received, the item is declared officially sold, and the corresponding seller and buyer are notified about the transaction.

We can depict the scenario as the following chain of operations (events) involving a seller, a buyer, and the manager:

\[
\begin{align*}
&\text{Put\_new\_item}(\text{Seller}) \rightarrow \text{Get\_new\_item}(\text{Manager}) \rightarrow \text{Make\_bid}(\text{Buyer}) \\
&\rightarrow \text{Take\_bid}(\text{Manager}) \rightarrow \text{Declare\_winner}(\text{Manager}) \rightarrow \text{Take\_bid}(\text{Manager}) \\
&\rightarrow \text{Make\_payment}(\text{Buyer}) \rightarrow \text{Selling\_confirmed}(\text{Seller}) \rightarrow \text{Item\_sold}(\text{Manager}) \\
&\rightarrow \text{Receive\_payment}(\text{Manager}) \rightarrow \text{Buying\_confirmed}(\text{Buyer}) \\
&\rightarrow \text{Item\_sold}(\text{Manager}) \rightarrow \text{Selling\_confirmed}(\text{Seller}) \rightarrow \text{Buying\_confirmed}(\text{Buyer})
\end{align*}
\]

We specify all these steps as the corresponding events in our Event-B model of the auction system. Each event also has an annotation indicating to which component it belongs. Moreover, we distribute the program variables among the involved components as well. Most variables are associated with the manager, except for `buyer\_log` and `seller\_log`, which belong to a (collective) seller and a (collective) buyer respectively.

\[
\begin{align*}
\text{SYSTEM \ Auction} & \quad \text{SEES \ Auction\_Context} \\
\text{VARIABLES} & \quad \text{bids, bids\_left, winner, paid\_items, item\_seller, buyer\_log, seller\_log, done} \\
\text{INVARIANT} & \quad \text{buyer\_log} \subseteq \text{paid\_items} \land \text{seller\_log} \subseteq \text{paid\_items} \\
& \quad \forall \text{ ii, ii} \in \text{ITEM} \land \text{done}(\text{ii}) = \text{TRUE} \Rightarrow (\text{ii} \in \text{buyer\_log} \Leftrightarrow \text{ii} \in \text{seller\_log}) \\
\text{INITIALISATION} & \quad \text{...} \\
\text{EVENTS} & \quad \text{...} \\
\text{Put\_new\_item} = \ldots \quad \text{/* seller */} & \quad \text{Make\_payment} = \ldots \quad \text{/* buyer */} \\
\text{Get\_new\_item} = \ldots \quad \text{/* manager */} & \quad \text{Receive\_payment} = \ldots \quad \text{/* manager */} \\
\text{Make\_bid} = \ldots \quad \text{/* buyer */} & \quad \text{Item\_sold} = \ldots \quad \text{/* manager */} \\
\text{Take\_bid} = \ldots \quad \text{/* manager */} & \quad \text{Selling\_confirmed} = \ldots \quad \text{/* seller */} \\
\text{Declare\_winner} = \ldots \quad \text{/* manager */} & \quad \text{Buying\_confirmed} = \ldots \quad \text{/* buyer */} \\
\text{END} & \quad \text{...}
\end{align*}
\]
A Contract-Based Approach to Ensuring Component Interoperability in Event-B

For brevity, in the invariant part, we show only a couple of the most important correctness properties of the auction system: (i) all bought and sold items should be paid for, and (ii) once the auction is done for a particular item, it should be recorded in both buyer’s and seller’s logs.

Below, we show two auction events in detail.

\begin{verbatim}
Get_new_item =  
     any ss, ii where  
     ss ∈ SELLER  
     ii ∈ ITEM  
     done(ii) = FALSE  
then  
     bids(ii) := (NO_BUYER → 0)  
     bids_left(ii) := MAX_BIDS  
     item_seller(ii) := ss  
end

Receive_payment =  
     any bb, ii where  
     bb ∈ BUYER  
     ii ∈ ITEM  
     bb = winner(ii)  
     ii ∉ paid_items  
then  
     paid_items := paid_items ∪ {ii}  
end
\end{verbatim}

The event Get_new_item specifies the manager’s reaction after getting a new item to sell. As a result, the bidding process is initiated. The event Receive_payment models recording of the received buyer payment by the manager.

In the initial model, the involved components can directly read each other’s state in the event guards. In the first refinement step, we decouple these components by explicitly introducing communication between them. The components communicate by sending and receiving requests. The received requests are stored in their input buffers, while the requests to be sent are put in their output buffers. The system model becomes more decentralised. However, the buyer and seller buffers are still collectively modelled by the corresponding arrays variables.

After an introduction of communication, the manager events Get_new_item and Receive_payment receive the following form:

\begin{verbatim}
Get_new_item =  
     any rr, ss, ii where  
     rr ∈ Selling_Req  
     rr ∈ manager_input  
     ss = Seller(rr)  
     ii = Item(rr)  
     done(ii) = FALSE  
then  
     bids(ii) := (NO_BUYER → 0)  
     bids_left(ii) := MAX_BIDS  
     item_seller(ii) := ss  
end

Receive_payment =  
     any bb, ii where  
     bb ∈ BUYER  
     ii ∈ ITEM  
     bb = winner(ii)  
     ii ∉ paid_items  
then  
     paid_items := paid_items ∪ {ii}  
end
\end{verbatim}

The first event now models the manager reaction on the received selling request in its input buffer, while the second event creates a payment confirmation request to be sent to the item seller and places it into the output buffer.

In addition, the refined model contains the events, sole purpose of which is to transport requests from the output buffer of one component to the input
buffer of the recipient component. Essentially, these events specify the behaviour of middleware that is responsible for implementing component communication.

We specify the middleware events that represent communication between the sellers and the managers as follows:

\[
\text{Seller to Manager} = \begin{cases} 
\text{any } ss, rr & \text{where } ss \in \text{SELLER} \\
ss \in \text{SELLER} & rr \in \text{seller\_output}(ss) \\
nr \notin \text{manager\_input} & \text{then} \\
\text{seller\_output}(ss) := \text{seller\_output}(ss) \setminus \{rr\} \\
\text{manager\_input} := \text{manager\_input} \cup \{rr\} \\
\end{cases}
\]

\[
\text{Manager to Seller} = \begin{cases} 
\text{any } ss, rr & \text{where } ss \in \text{SELLER} \\
ss \in \text{SELLER} & rr \notin \text{seller\_input}(ss) \\
nr \in \text{manager\_output} & \text{then} \\
\text{seller\_output}(ss) := \text{seller\_output}(ss) \cup \{rr\} \\
\text{manager\_input} := \text{manager\_input} \setminus \{rr\} \\
\end{cases}
\]

The respective events for the buyer-manager communication are added as well.

In the second refinement step, we decompose the system specification by explicitly introducing the manager component as well as the buyer and seller components for each element of the given sets \text{BUYER} and \text{SELLER}. We will rely on the modularisation extension of Event-B to decompose the system model. The system model is refined into that of the middleware, which calls, when needed, operations from the corresponding introduced components.

To perform such a decomposition refinement step, we need to

1. Define separate module interfaces for the manager, a buyer and a seller;
2. Distribute the system state (not belonging to the middleware) among the introduced components;
3. In each interface, define the callable operations for accessing the component input and output buffers;
4. Distribute the events describing the autonomous component behaviour among the components, defining them as the corresponding module processes;
5. Define the gluing invariants, relating the array variables modelling the collective knowledge (e.g., buyer and seller logs) with the respective variables belonging to single module instances. Essentially, this allows us to propagate system-level properties into the definition of the derived module interfaces.

In Fig.1.5, we present the defined interface for the seller component. In a similar way, the interfaces for the manager, buyer, and seller components are defined. Having the seller interface defined, it is rather straightforward to obtain the corresponding class contract for all sellers (see Fig.1.6).

The core refined model now consists only of the state and events of the system middleware. The additional clause \text{USES} creates one instance of the manager component as well as a number of instances of the seller and buyer
A Contract-Based Approach to Ensuring Component Interoperability in Event-B

**INTERFACE** Seller\( (id) \)
**VARIABLES** input, output, log

**INVARIANT** \( (\forall rr \in output. \text{Item}(rr) \notin \log) \land (\forall rr \in input. \text{Id}(rr) = id) \land \\
(\forall rr \in input \cap \text{Selling_Req}. \text{Item}(rr) \notin \log) \land \ldots \)

**INITIALISATION** \( \text{input} = \emptyset \land \text{output} = \emptyset \land \text{log} = \emptyset \)

**PROCESS**
- \( \text{Put_new_item} = \text{any } ii \text{ where } ii \in \text{ITEM} \land ii \notin \log \text{ then } \text{output} := \text{output} \cup \{\text{Sell_Request}(id \mapsto ii)\} \text{ end} \)
- \( \text{Selling_confirmed} = \text{any } rr \text{ where } rr \in \text{input} \cap \text{Pay_Confirmation} \text{ then } \text{log} := \text{log} \cup \{ii\} \parallel \text{input} := \text{input}\{rr\} \text{ end} \)

**OPERATIONS**
- \( \text{Add_request} = \text{any } rr \text{ pre } rr \in \text{Pay_Confirmation} \land rr \notin \text{input} \land \text{Id}(rr) = id \text{ post } \text{input}' = \text{input} \cup \{rr\} \text{ end} \)
- \( \text{Take_request} = \text{any } rr \text{ pre } rr \in \text{Selling_Req} \cup \text{Selling_Ackn} \land rr \in \text{output} \text{ post } \text{output}' = \text{output}\{rr\} \text{ end} \)

**END**

**FIGURE 1.5**: Interface Component

components – one per each element of the given sets \( \text{SELLER} \) and \( \text{BUYER} \).

**SYSTEM** Auction\_2nd\_refinement
**USES** \( \text{Manager as manager, Seller[SELLER], Buyer[BUYER]} \)

**VARIABLES** ...

**INVARIANT** ...

**INITIALISATION** ...

**EVENTS**
- \( \text{Seller_to_Manager} = /\* \text{middleware } /\* \)
- \( \text{Manager_to_Seller} = /\* \text{middleware } /\* \)
- \( \text{Buyer_to_Manager} = /\* \text{middleware } /\* \)
- \( \text{Manager_to_Buyer} = /\* \text{middleware } /\* \)

**END**

The invariant clause must also include the gluing invariants between the abstract model variables and the variables of the decentralised system, e.g.,

\[ \forall ss, ss \in \text{SELLER} \Rightarrow \text{input}(ss) = \text{seller_input}(ss) \]
\[ \forall ss, ss \in \text{SELLER} \Rightarrow \text{log}(ss) = \text{seller_log}(ss) \]

In other words, these invariants relate the array variables of a centralised system before the refinement step, \( \text{seller_input} \) and \( \text{seller_log} \), with the corresponding variables, \( \text{input} \) and \( \text{log} \), of the Seller components.

Below we show in detail a couple of middleware events. Note that now they are defined only in terms of the operation calls to the interface operations of the involved components.
COMPONENT CLASS Seller(id)

VARIABLES input, output, log

INVARIANT (∀rr ∈ output. Item(rr) ∉ log) ∧ (∀rr ∈ input. Id(rr) = id) ∧ (∀rr ∈ input ∩ Selling_Req. Item(rr) ∉ log) ∧ ···

INITIALISATION input = ∅ ∧ output = ∅ ∧ log = ∅

ACTIONS

Put_new_item = params ii pre
ii ∈ ITEM ∧ ii ∉ log
post
output′ = output ∪ {Sell_Request(id → ii)}
end

Selling_confirmed = params rr pre
rr ∈ input ∩ Pay_Confirmation ∧ Id(rr) = id ∧ Item(rr) ∉ log
post
log′ = log ∪ {ii} ∧ input′ = input\{rr\}
end

Add_request = params rr pre
rr ∈ Pay_Confirmation ∧ rr ∉ input ∧ Id(rr) = id
post
input′ = input ∪ {rr}
end

Take_request = params rr pre
rr ∈ Selling_Req ∪ Selling_Ackn ∧ rr ∈ output
post
output′ = output\{rr\}
end

END

FIGURE 1.6: The Seller class contract

Seller_to_Manager =
any ss, rr where
ss ∈ SELLER
rr ∈ output(ss)
rr ∉ man.input
then
Take_request(ss)(rr)
manager_Add_request(rr)
end

Manager_to_Seller =
any ss, rr where
ss ∈ SELLER
rr ∉ input(ss)
rr ∈ man.output
then
manager_Take_request(rr)
Add_request(ss)(rr)
end

The derived interfaces of the buyer and the seller modules represent their contracts. The auction system can be developed by composing the seller, buyer and middleware components. To ensure interoperability of the implemented components, the designers need to verify that the components comply to the contracts defined above.

While developing the auction system, we have employed the strategy that is usually used for the development of distributed systems by refinement [15, 14, 2]. Namely, we start from an abstract centralised system model. In such a model, components can directly access each other’s state. This allows us to simplify definition of the system-level invariant properties, including the ones that express interoperability conditions. In a number of refinement steps, we introduce a representation of the detailed functional requirements as well
as communication mechanisms between the components. Finally, we perform model decomposition and arrive at a decentralised system model. Such a model typically consists of the components that communicate with each other via the communication support provided by the introduced middleware component. The decomposition refinement step also allows us to derive the corresponding contracts of the system components.

1.5 Conclusions

In this paper, we have presented a rigorous approach to ensuring component interoperability. The main idea of the approach was to derive contracts of the constituting components from the overall system specification in Event-B. The modularisation extension of Event-B allowed us to decompose the system specification into the independent components – modules. A module interface defines a family of components that might vary in implementation and the internal behaviour but nevertheless comply to the contract defined by the module interface. As a next step, we have demonstrated how to transform such a module interface into the corresponding contract of a component.

Our ideas were illustrated by an example – an auction system. An application of the proposed approach allowed us to formally define interoperability of the buyer, seller and middleware components, i.e., ensure their correct interactions during the auction activities.

Our approach has been inspired by the seminal works of Kaisa Sere and her group on refinement, modularisation and decomposition in the action systems formalism. The fundamental theoretical aspects of the refinement approach to the development of distributed systems were presented by Back and Sere in [4]. The modularisation idea for the action systems framework was proposed in [5]. The systems approach to development by refinement that enables propagation of the system-level properties into the models of components have been proposed in [7]. Since Event-B has adopted many aspects of the theoretical foundations of the action systems framework, in our approach we followed the decomposition-based development approach defined by the action systems framework. We extended the original ideas by the notions of module interface and component contracts.

The concept of contracts has been exploited in various domains and development approaches. Majority of the approaches aim at facilitating compositional component-based development. A foundational work on rigorous aspects of such development was done within the EU FP6 SPEEDS project [9]. The project studied theoretical aspects of formal component-based design. It proposed a component-based framework that defines component properties as extended transition systems and different compositional approaches associated with it.
A contract-based top-down design methodology was proposed by Quinton and Graf [18]. This work explored different forms of conformance of a component to a contract that is defined by corresponding notions of refinement. The authors also define refinement relation between contracts. There is also a vast body of research on contracts that is based on behavioural typing, where a contract is an abstraction of the component behaviour as a transition system, see, e.g., [6, 8]. As a result of this research, many contract languages were proposed, which define various ways contracts can be composed and compared, while the behavioural subtyping relation, which can be seen as refinement, is used for compliance of the component behaviour to a contract.

In our work, we pursued another goal – we aimed at deriving contracts from a refined and decomposed system specification in Event-B. The refinement process allows us to preserve the global system properties, while gradually decoupling the components and formally defining their interfaces. As a result, the derived contracts ensure that any component conforming to its contract is interoperable with the other components of the system.

Defining contracts for complex component-based systems is still a challenging problem. We believe that the proposed approach helps to alleviate it. Indeed, it allows us to derive component contracts from a specification that rigorously defines the overall interoperability conditions for components and their environments. As a future work, we are planning to experiment with the notion of probabilistic contracts, in particular, in the domain of fault tolerant systems.
Bibliography


Bibliography


