JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 1, No. 3
Special issue: TOOLS USA 2002 proceedings

A logic-based formalization for component specification

Juliana Küster Filipe,
LFCS, Division of Informatics, University of Edinburgh, United Kingdom

We consider a component-based approach to modelling complex systems using UML. We describe how component concepts at a specification level (interfaces, components, architectures) can be formalized in a uniform way using a distributed logical framework. In the logic MDTL, each component has associated to it a local logic consisting of a home logic and a communication logic. Component contracts are captured by formulae in MDTL. In particular, a clear distinction between usage and realization contracts is made: the former is captured as formulae in the home logic of the interface specification, whilst the latter is expressed by formulae in the communication logic of the component specification. Moreover, we are investigating an extension of the framework for expressing dependability requirements.

1 INTRODUCTION

As part of an interdisciplinary research project focusing on the specification and design of complex and highly dependable systems, we are interested in how formal models can be used to analyse dependability requirements and consequently aid designers understand such systems. In this paper, we consider a component-based approach to system modelling, and describe how to formalize component concepts using a distributed logical framework.

Component-based software development is an emerging field with promising solutions for dealing with the rapidly changing requirements of present-day software applications (see e.g., [12]). While component technologies such as COM+ and Enterprise JavaBeans are becoming widely used, components lack an adequate treatment at the specification level. Component specification is essential: it is not possible to manage change, substitution and composition of components successfully if components have not been specified properly.

UML [9] has become a popular modelling language to describe systems following an object-oriented approach. UML is mainly a diagrammatic language offering several diagrams to capture different aspects of a system. It also includes the Object Constraint Language (OCL), a textual notation for representing static constraints

Cite this article as follows: Juliana Küster Filipe: *A logic-based formalization for component specification*, in Journal of Object Technology, vol. 1, no. 3, Special issue: TOOLS USA 2002 proceedings, pages 231–248, http://www.jot.fm/issues/issue_2002_08/article13

on the model which cannot be given through the diagrams.

UML also offers a component concept. However, UML has not been developed with the aim of allowing a component-oriented style of software development. Components in UML are low level units that exist at runtime, and thus do not denote main features for a conceptual or specification level.

In [2], Cheesman and Daniels define a pragmatic extension of UML to capture important component concepts such as component specification, component interfaces, component implementations, and component objects. In particular, they apply a *design by contract* approach to components; whilst components provide services, they may require services from other components as well. The provide/require dependencies between components are described by contracts. Component contracts are represented partially declaratively (using OCL) and partially operationally (using UML collaboration diagrams). OCL is used for describing pre- and postconditions of interface operations. Collaboration diagrams are used for capturing component interactions.

We adopt the approach in [2] for modelling component-based systems using UML though using a declarative description of component contracts. We use a *Catalysis* [4] like notation where OCL lacks expressiveness to describe component contracts as needed. We show how the distributed logic MDTL and the framework developed in [7, 8] can be used to formalize components and their contracts at a specification level. The importance of such a formalization lies in the development of verification tools which could be used, for instance, to check component contracts and whether components can be combined in a useful way.

For illustrating our approach, we are going to model part of the ParcelCall system. ParcelCall[1] is an European research and technology development project looking at creating a *parcel localization system*: an open distributed system which is to be integrated with the legacy systems of transport and logistic companies. This case study also motivates the need for a future extension of our logical framework towards an interdisciplinary approach to the development of dependable computer-based systems. We shall discuss this extension briefly.

The paper is structured as follows. The next section describes component concepts for specification and how they are represented. Section 3 introduces ParcelCall, and models particular aspects of the components and their interactions. The logical framework is described in Section 4, and used to formalize component concepts in Section 5. The paper finishes with a discussion on an extension of the logical framework for dependability and some concluding remarks in Section 6.

[1]Publications and project description can be found at http://www.parcelcall.com.

2 BASIC CONCEPTS

In this paper component concepts are understood for specification essentially as given in [2]. The following defines some component concepts at a specification level.

Components are units of software that provide services to other components and possibly require services from other components. The provide/require dependencies between components are described by contracts. They denote quite different kinds of contracts and are thus in [2] specified separately: the *usage contract* denotes a provide dependency whereas the *realization contract* denotes a require dependency.

The usage contract is specified as a *component interface*. A component interface defines the details of a contract with its clients. It defines the operations it provides, what their signatures are, their effects and when these effects can be guaranteed to hold. To be able to specify the operations in an interface it is necessary to partially know the state of the component. Consequently, the interface specification contains a so-called *information model* with attributes, associations and classes as needed. The information model denotes a view of the state model of the component.

By contrast, the realization contract is defined within the *component specification* and describes some details relevant to the implementation of the component. A component specification consists of a set of offered interface specifications together with a definition of how these interfaces are related (that is, how the corresponding information models match each other); and a set of required interfaces from other components. Finally, it may contain further constraints such as on the implementation of an operation (given through component interactions). Structural and behavioural dependencies between component specifications are captured by the *component specification architecture*.

[2] provides a process for component-based software development focusing on the specification phase: the identification of the components, the definition of their dependencies, the construction of their specifications, and finally the production of flexible application architectures. It shows first, prior to specification, how to represent requirements in such a way that it eases the construction of component specifications, and then how to create the component specifications themselves.

The process described in [2] uses UML1.3 for modelling component specification. It provides a pragmatic extension of UML to capture the above concepts. In the initial requirements phase two models are mainly used: a class diagram to represent the initial business concept model, and a use case diagram to capture user-system interactions. These provide the basis for identifying required interface operations as well as the component dependencies.

The specification contains four main artifacts: a business type model, interface specifications, component specifications and component architecture. All artifacts are modelled using class diagrams, except the component interactions within the component architecture, which are modelled operationally by a collaboration diagram. At this point we diverge from [2] and describe component interactions

declaratively using a notation similar to that taken in the *Catalysis* approach [4]. Because dependencies between components and consequently their contracts are essential in a component-based approach and to be able to manage change of such contracts and/or components, we need a precise description of them. Notice that usage contracts are already described using OCL in [2], however further dependencies or interactions between components (forming the realization contract) are not, because this is not possible in OCL1.x. Indeed, OCL2.0 aims at including a notation which would allow one to express components receiving events and reacting to them (see [6]). Such an extension could replace our *Catalysis*-based notation (which we introduce merely for convenience) and can in any case be formalized in our logical framework.

The above concepts and their representation in UML is given in Section 3 as needed when we model part of our ParcelCall example.

3 THE PARCELCALL EXAMPLE

The ParcelCall project explores the development of a low cost information infrastructure that will enable the continuous information about the exact geographic position of parcels at any time. Logistic or transportation companies (referred as carriers) will then be able to offer an additional service to customers: a customer can query the location and status of her transportation goods.

The ParcelCall system has three main components:

- a *Mobile Logistic Server* (MLS): is an exchange point or a transport unit (container, trailer, freight wagon, etc). The transport units carry the parcels. Since containers can be inside other containers MLSs form a hierarchy. MLSs always know their current location via the GPS satellite positioning system.
- a *Goods Tracing Server* (GTS): comprises several databases which contains MLS hierarchies. Moreover, it keeps track of all the parcels registered in the ParcelCall system. GTS is also the component which is integrated with the legacy system of transport or logistic companies.
- the *Goods Information Server* (GIS): is the component which interacts with the customers and provides the authorised customer the current location of her parcels, keeps her informed in case of delivery delays, etc.

For the purpose of this paper we will not address the integration of ParcelCall with a carrier system nor the arising dependability issues (see Section 6 for a discussion). We show how a component-based approach can be used for modelling the ParcelCall system. We focus on some of the interactions between the components within the localization system triggered by a request to localize a parcel. We use UML to specify some interface and component specifications as well as the component specification architecture of ParcelCall.

In [2] the stereotypes <<comp spec>> and <<interface type>> are introduced to describe component specifications and interfaces respectively. The UML lollipop notation for interfaces is sometimes also used for the interface types when describing the system architecture. Figure 1 shows the architecture of ParcelCall: the three main components, some of their offered interfaces and component dependencies. For example, the GIS component offers two interfaces ILocalizeParcel and IDispatchParcel, and requires a service from component GTS via the interface IParcelInfo. We also show where the legacy system from the carrier is going to be integrated with ParcelCall.

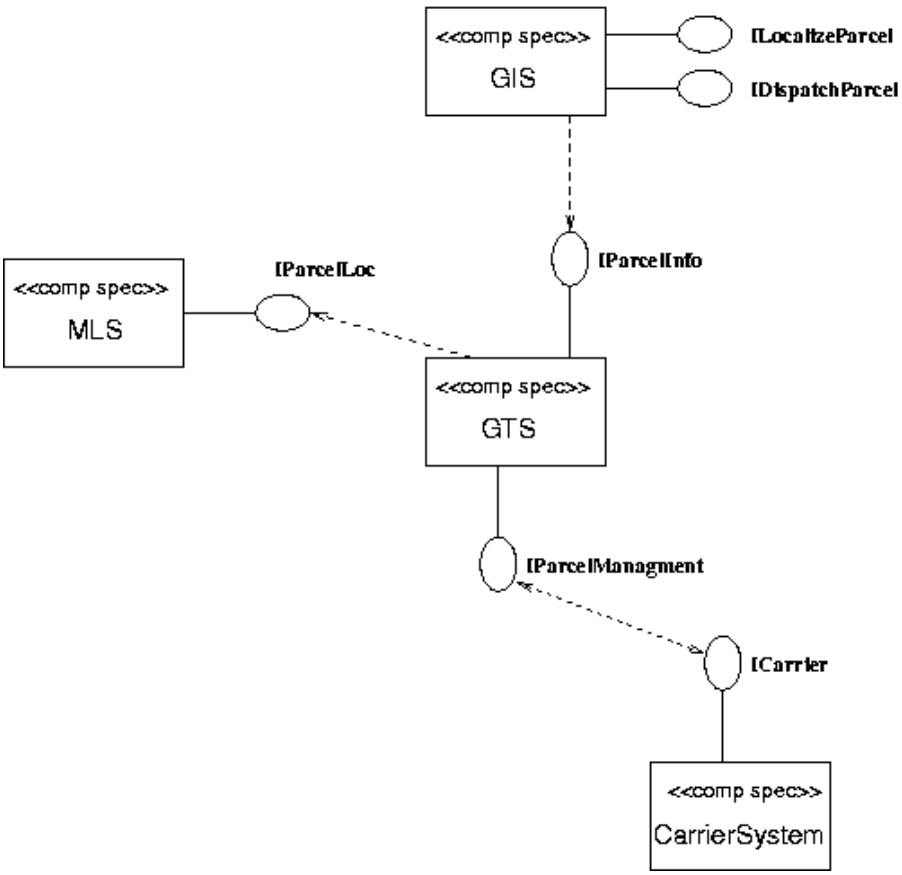


Figure 1: ParcelCall component specification architecture.

The interfaces from the GIS component will establish communication with customers: for instance a customer can enter a request to find out the current location of a parcel via the ILocalizeParcel. Figure 2 shows the interface ILocalizeParcel with the required information model and one operation.

The operation `whereParcel` is a query and returns the current location of a certain parcel provided the parcel exists in the system and the customer is authorised to know its location (we assume a data type `Coordinates`). The interface specification therefore contains the following OCL precondition for the operation.

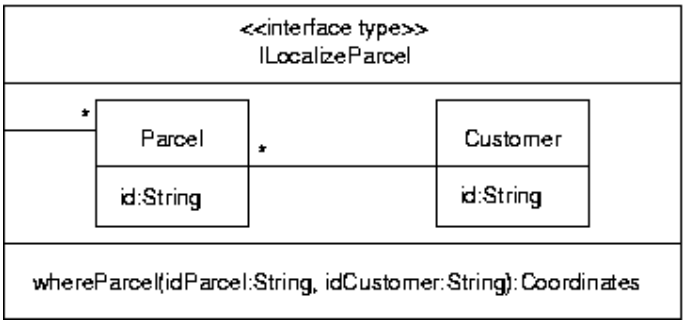


Figure 2: The interface **ILocalizeParcel**.

```
context ILocalizeParcel::whereParcel(idParcel:String,
                                     idCustomer:String):Coordinates
pre: self.Parcel->exists(p| p.id=idParcel and p.Customer=idCustomer)
```

What is not explicit to a client of GIS is that to provide the parcel location GIS requires a service from component GTS. The specification of GIS contains therefore the two offered interface specifications, the required interface specification IParcelInfo and further constraints (below). Figure 3 shows the required interface IParcelInfo of component GTS.

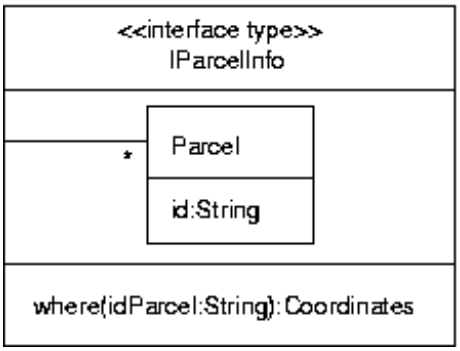
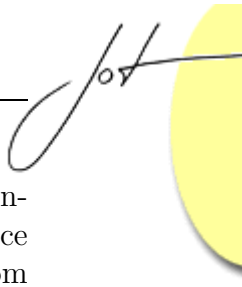


Figure 3: The interface **IParcelInfo**.

Within GIS constraints can be imposed on the implementation of the operation `whereParcel`. We provide two possible constraints written in *Catalysis*-based notation (see [4] for details). The first possible constraint is as follows.

```
context GIS::whereParcel(idParcel:String,idCustomer:String):Coordinates
post: [[r:=-> IParcelInfo.where(idParcel)]]  and result = r
```



Here the postcondition tells us that whereParcel's execution *synchronously* invokes another operation. A message is sent synchronously to an arbitrary instance of the component supporting the interface IParcelInfo. r is the value returned from the message and the result value of whereParcel is equal to r.

Alternatively, we can give another constraint, involving *asynchronous* invokation.

```
context GIS::whereParcel(idParcel:String,idCustomer:String):Coordinates
post:[[ sent m-> IParcelInfo.where(idParcel)]]
                                    and [[ m(idParcel)=r ]]  and result = r
```

Here, executing whereParcel will asynchronously send a message to an arbitrary instance of the component supporting the interface IParcelInfo. m is an event identifier used also to describe that when the sent message has eventually been completed it returns r. Finally, the result of whereParcel is equal to r.

The component GTS will require a service from component MLS to be able to satisfy the service required by GIS. We omit further details.

4 LOGICAL FRAMEWORK

In what follows we present the logical framework developed in [7, 8]. The presentation of the formalism has been simplified in this paper to increase readability. We refer the interested reader to [7] for further details. Note that in [7, 8], components for specification have been designated *object-oriented modules* or *modules* for short. In particular, MDTL stands for Module Distributed Temporal Logic. For consistency, however, we stick to the component terminology in this paper.

The Idea

In our framework, component descriptions are theory presentations of a certain logic. A component description is a pair consisting of a *component signature*, defining the specific vocabulary symbols that are relevant for the description of the component; and a set of *component axioms*, a collection of formulae in the logic generated from the signature. The logic that has been developed for describing components is MDTL, a distributed temporal logic which is interpreted over labelled prime event structures. The idea of the distributed approach is that each distributed component in a system has its own *local* logic whereby this is split into a *home* logic and a *communication* logic. The home logic allows one to express internal component properties, whereas the communication logic expresses interactions with other components.

Note that in our framework a component is understood as a collection of interacting object classes, unlike in [2] where a component is essentially a class and its instances complex objects. We return to this distinction in Section 5. Further, in

our framework methods or operations are called actions, and associations are covered implicitly through attributes.

Component Signature

We recall the definition of an order-sorted data signature $\Sigma_D = (S_D, \Omega_D, \leq_D)$, where S_D is a finite set of *data sorts*; Ω_D is an $S_D^* \times S_D$-indexed family of sets of *data operations*, and $\leq_D \subseteq S_D \times S_D$ is a *binary relation* on the set of data sorts such that $(S_D, \leq_D)$ is a partial order. For each $o \in \Omega_{D w,s}$, w is the *parameter list* of the operation o and s the *result sort*. The elements of $\Omega_{D \varepsilon,s}$ are called *constant symbols* of sort s. The partial order on the data sorts specifies a subsort relation. Moreover, the partial order is monotone, that is, it satisfies the following condition: if $o \in \Omega_{D s_1 \ldots s_n,s} \cap \Omega_{D r_1 \ldots r_n,r}$ and $s_i \leq_D r_i$ for $1 \leq i \leq n$ then $s \leq_D r$. This condition on the operations allows one to deal with partial functions, overloading and polymorphism. It is particularly important for expressing inheritance and polymorphism in object-oriented languages.

Let X be an S_D-indexed family of disjoint sets of *variables*. A data signature may be extended with variables by considering them as constant symbols of a given sort. A data signature with variables is sometimes written $\Sigma_D(X)$. From the symbols defined in the order-sorted data signature and the variables we can construct *data terms* in the usual way. $T_{\Sigma_D,s}(X)$ denotes the set of data terms of sort s over $\Sigma_D(X)$. $T_{\Sigma_D}(\emptyset)$ is the family of closed terms, also written T_{Σ_D}. Terms denote a certain value, so they can be evaluated under a given interpretation. We refer the interested reader to [5] for a detailed presentation of order-sorted signatures, their interpretation structures (algebras), and categorical results. As an example of the latter, we can define morphisms between order-sorted signatures in such a way that the signatures and morphisms define a category. For the purpose of this paper it suffices to understand a morphism between signatures as a function mapping the symbols of one signature onto the symbols of another. We omit further details in this paper. We want to define the notion of a component signature using order-sorted signatures so that we can profit from all the known results on them.

A component is more than a class and typically contains several related classes. For describing a component signature, apart from data sorts and data operations as above, we will need *object* and *component* sorts (S_O and S_C) and operations on them (Ω_O and Ω_C). Intuitively, each class is equipped with an object sort. Since classes can be arranged in hierarchies through inheritance, a partial order defined on the object sorts reflects an inheritance relation. Furthermore, a partial order on component sorts denotes component dependency (also referred as subcomponent relationship).

A class describes the *attributes* and *actions* of its potential *instances*. Attributes, actions, and instances can be understood as special object operations, but we need to be able to distinguish them: we need to know whether a certain object operation is an action operation or else. Thus, we distinguish between: an *attribute object sort* (S_O^{at}),

an *action object sort* (S_O^{ac}), and an *instance object sort* (S_O^i). We also distinguish within action object sorts between *synchronous* action sorts (S_O^{syn}), *asynchronous send* action sorts (S_O^{asd}), and *asynchronous receive* action sorts (S_O^{arc}), all disjoint sets.

The following defines what we call a component kernel signature.

Definition 4.1 (Kernel Signature) *A* kernel signature *is an order-sorted signature* $\Sigma = (S, \Omega, \leq)$ *such that:*

- S *is a finite set of sorts,* data, object *and* component *sorts, that is,* $S = S_D \cup S_O \cup S_C$ *where:*
  - $S_O = S_O^i \cup S_O^{at} \cup S_O^{ac}$ *is a disjoint union of sets of object sorts such that* $S_O^{ac} = S_O^{syn} \cup S_O^{asd} \cup S_O^{arc}$ *is a disjoint union as well.*
  - $S_C = S_C^e \cup S_C^i$ *is a union of sets of component sorts such that* $S_C^e \cap S_C^i = \{\alpha\}$. α *is designated the* local *component sort, the sorts in* S_C^e *are* export *component sorts, and in* S_C^i *are* import *component sorts.*
- Ω *is an* $S^* \times S$*-indexed family of sets of operation symbols such that* $\Omega_D \subseteq \Omega$. *Let* $S^i = S_O^i \cup S_D$. *Further operations in* Ω *are*
  - $\Omega_{s^i x^i, s^i}$ *with* $x^i \in S^{i*}$ *and* $s^i \in S_O^i$*, denolte object instance operations;*
  - $\Omega_{s^i s^{at}, r^i}$ *with* $s^i \in S_O^i$, $s^{at} \in S_O^{at}$ *and* $r^i \in S^i$*, denote attribute operations;*
  - $\Omega_{s^i x^i, s^{ac}}$ *with* $s^i \in S_O^i$, $x \in S^{i*}$ *and* $s^{ac} \in S_O^{ac}$*, denote action operations;*
  - $\Omega_{\varepsilon, c}$ *with* $c \in S_C$ *is a singleton, denotes a component instance operation.*
- $\leq \subseteq S \times S$ *is a binary relation on the set of sorts such that* $(S, \leq)$ *is a partial order satisfying:*
  - *only sorts of the same kind can be related;*
  - *for any* $s_1, s_2 \in S_O$, $s_1^i \leq s_2^i$ *iff* $s_1^{at} \leq s_2^{at}$ *iff* $s_1^{ac} \leq s_2^{ac}$*;*
  - *for any* $c \in S_C$, $c \leq \alpha$*;*
  - *for any* $c, d \in S_C \setminus \{\alpha\}$, *if* $c \neq d$ *then* $c \not\leq d$ *and* $d \not\leq c$.

  The monotonicity condition given before has to hold.

Attribute operations and action operations are always associated with an object instance sort. For example, the attribute operation $o \in \Omega_{s^i s^{at}, r^i}$ is associated to an object instance sort given by s^i. (We antecipate that this is to be able to build terms in a special way.) Moreover, s^{at} indicates that the operation is an attribute operation and r^i is the sort of the attribute. For an action operation $m \in \Omega_{s^i x^i, s^{ac}}$, x^i denotes

the sorts of the arguments including possibly the result sort. The reason why in an instance operation $o \in \Omega_{s^i x^i, s^i}$, s^i appears twice is so that instance operations can be inherited by subclasses: notice that if the first s^i is omitted the monotonicity condition can never be satisfied. The only component operations available are component instance operations. We need a unique constant component operation that denotes the instance of the component at hand.

From a kernel signature we can construct not only data terms (as usual) but instance, attribute and action terms for objects, and component terms. For a given signature $\Sigma(X)$, we will denote $T_\Sigma(X)$ the family of sets of data and instance terms, $ATT_\Sigma(X)$ the family of sets of attribute terms, and $ACT_\Sigma(X)$ the family of sets of action terms. Since action sorts are distinguished we also have synchronous action terms $S_\Sigma(X)$, asynchronous send action terms $SAC_\Sigma(X)$ and asynchronous receive action terms $RAC_\Sigma(X)$. C_Σ denote component instance terms.

Instance terms are constructed like data terms ignoring the first argument sort. Attribute terms have the form $t.a$ where t is an instance term and a an attribute operation. The sort of the term $t.a$ is given by the result sort of a, i.e., if $a \in \Omega_{s^i\ s^{at}, r}$ then the attribute term $t.a$ has sort r and t is a term of sort s^i. Action terms have the general form $t.c(t_1, \ldots, t_n)$ where t is an instance term, c denotes an action operation, and there is a list (possibly empty) of argument terms $t_1 \ldots t_n$. The sort of an action term $t.c(t_1, \ldots, t_n)$ is given by the action sort of c, i.e., if $c \in \Omega_{s^i x, s^{ac}}$ then the action term is of sort s^{ac}. Component instance terms, or component terms for short, are always closed and given by constants of a given sort $m \in S_C$ since the only component operations available are constants. In particular, $c \in \Omega_{\varepsilon,\alpha}$ is the *local* component term. Closed data and instance, attribute, and action terms are written T_Σ, ATT_Σ and ACT_Σ respectively.

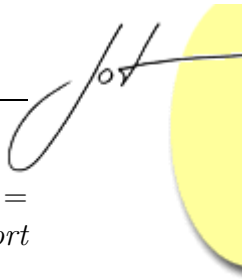
The interpretation structures over kernel signatures are essentially the same as for order-sorted signatures. Morphisms between kernel signatures can be defined in a similar way as well.

An export signature over a kernel signature is defined as follows.

Definition 4.2 (Export Signature) *Let Σ_1 and Σ_2 be kernel signatures, and $\mu : \Sigma_2 \hookrightarrow \Sigma_1$ be an inclusion morphism. Let α_1 and α_2 be the local component sorts of Σ_1 and Σ_2 respectively. $E = (\Sigma_2, \mu)$ is an* export signature *over Σ_1 iff $\alpha_2 \in S^e_{C1}$ and $S_{C2} = \{\alpha_2\}$. Moreover, for two arbitrary export signatures E_1 and E_2 over Σ with β_1 and β_2 the local component sorts, $E_1 \neq E_2$ iff $\beta_1 \neq \beta_2$.*

The local component sort of the kernel of an export signature over Σ has to be an export component sort of Σ. An export signature has a unique component sort.

Import signatures are defined in exactly the same way (just replace in the above definition S^e_{C1} by S^i_{C1}). With the notions of kernel, export and import signatures we can define a component signature.



Definition 4.3 (Component Signature) *A* component signature *is a triple* $\Theta = (\Sigma, Imp, Exp)$ *where* Σ *is a kernel signature,* Imp *and* Exp *are finite sets of import and export signatures over* Σ *respectively.*

From an export signature over a kernel signature we can build a special kind of component signature which we call *view* component signature (of the component signature of the kernel). I.e., let $E = (\Sigma_1, \mu)$ be a an export signature over Σ, and Σ be the kernel signature of a component signature given by Θ. The component signature $\Theta_1 = (\Sigma_1, \emptyset, \{(\Sigma_1, id)\})$ is a *view* of the component signature Θ where id is an identity kernel morphism.

Distributed Logic

We now introduce a simplified component logic MDTL.

Definition 4.4 (MDTL) *Let* $\Theta = (\Sigma, Imp, Exp)$ *be a component signature,* $\beta \in S^i_C$*, and* Σ_β *be the corresponding kernel signature. Let* $\Sigma_\beta = (S_\beta, \Omega_\beta, \leq_\beta)$*, and* X *be an* S^i_β*-indexed family of sets of variables,* $x \in X_s$ *and* $s \in S^i_\beta$*. The abstract syntax of* MDTL_Θ *may be defined as follows:*

$$
\begin{aligned}
&\text{MDTL}_\Theta ::= \{\text{MDTL}_m\}_{m \in C_{\Sigma,\beta}} \\
&\text{MDTL}_m ::= m.H_m \mid m.C_m \\
&H_m ::= \text{ATOM}_m \mid \neg(H_m) \mid (H_m \Rightarrow H_m) \mid \forall_x(H_m) \mid (H_m \; \mathcal{U}_\forall \; H_m) \mid (H_m \; \mathcal{U}_\exists \; H_m) \mid \\
&\qquad (H_m \; \mathcal{S} \; H_m) \mid \Delta(H_m) \\
&C_m ::= \; SY_m \leftrightarrow k.SY_k \mid AS_m \rightarrow k.AR_k \mid AR_m \leftarrow k.AS_k \mid \forall_x(C_m) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{for some } k \in C_{\Sigma,\beta}, m \neq k \\
&SY_m ::= \odot S_{\Sigma_\beta}(X) \mid SY_m \wedge Q_m \mid \forall_x(SY_m) \mid \exists_x(SY_m) \\
&AS_m ::= \odot SAC_{\Sigma_\beta}(X) \mid AS_m \wedge Q_m \mid \forall_x(AS_m) \mid \exists_x(AS_m) \\
&AR_m ::= \odot RAC_{\Sigma_\beta}(X) \mid AR_m \wedge Q_m \mid \forall_x(AR_m) \mid \exists_x(AR_m) \\
&Q_m ::= \text{ATOM}_m \mid \neg(Q_m) \mid \forall_x(Q_m) \\
&\text{ATOM}_m ::= true \mid \triangleright ACT_{\Sigma_\beta}(X) \mid \odot ACT_{\Sigma_\beta}(X) \mid T_{\Sigma_{\beta,s}}(X) \; \theta \; T_{\Sigma_{\beta,s}}(X) \mid \\
&\qquad ATT_{\Sigma_{\beta,s}}(X) \; \theta \; T_{\Sigma_{\beta,s}}(X)
\end{aligned}
$$

Each component Θ has a component logic given by MDTL_Θ. MDTL_Θ associates to each imported subcomponent of Θ a local logic MDTL_m, where m is the component term of the subcomponent. The local logic MDTL_m allows m to make assertions about itself, and what it knows from communication with other subcomponents. The local logic MDTL_m is split into a component *home* logic H_m and a component *communication* logic C_m.

H_m is a first-order temporal logic with an additional operator Δ, the *concurrency* operator. The *atomic* formulae of the home logic H_m are given by ATOM_m. An atomic formula can be the logical constant *true*; the predicate $\triangleright$ (enabling) applied

to an action term; the predicate $\odot$ (occurrence) applied to an action term; or the predicate θ applied to two data terms or to an attribute and a data term, where θ is a comparison predicate (e.g., $=, \leq, \ldots$). Notice that the predicates $\rhd$ and $\odot$ are needed to be able to distinguish among actions that may occur next (are enabled) and are occurring. Essentially, they reflect pre and postconditions of actions.

Formulae in H_m can be obtained by applying successively the connectives $\neg$ and $\Rightarrow$, the temporal operators $\mathcal{U}$ (until) and $\mathcal{S}$ (since), the operator Δ and the $\forall$ quantifier to atomic formulae. Within the temporal operators we distinguish between a *for all until* $\mathcal{U}_\forall$, and an *exists until* $\mathcal{U}_\exists$. They are used to reflect the branching-time nature of the temporal logic. Moreover, this distinction is only sensible for the future-oriented temporal operators.

The logical constant $false$ and the well-known connectives of propositional calculus such as $\wedge$, $\vee$ and $\Leftrightarrow$ are defined in terms of $\neg$ and $\Rightarrow$ in the usual way, whereas $\exists$ can be obtained combining $\neg$ and $\forall$. Furthermore, the temporal operators *next* X, *sometime in the future* F, *always in the future* G, *yesterday* Y, *sometime in the past* P, and *always in the past* H can be derived from $\mathcal{U}$ and $\mathcal{S}$. Notice that our $\mathcal{U}$ and $\mathcal{S}$ are weak, they do not include "now". Details of the derivations can be found in [7].

The new operator Δ is a concurrency operator that can be used to express, for instance, that certain actions are executed concurrently. It is not used in this paper, so we dismiss it.

The communication logic C_m allows one to express communication among several objects from distinct subcomponents of Θ. A communication formula thus expresses intercomponent communication. Notice that intracomponent communication is expressed as a formula in the home logic instead.

A formula in the logic C_m reflects the knowledge the component denoted by m has of others, gained through communication, and from the local viewpoint of m. The component denoted by m may communicate with any other imported subcomponent of Θ denoted by k.

There are three possible statements in the logic concerning communication, the first refers to synchronous communication while the second and third refer to asynchronous communication.

$$\underbrace{SY_m \leftrightarrow k.SY_k}_{\text{synchronous}} \mid \overbrace{\underbrace{AS_m \rightarrow k.AR_k}_{\text{send}} \mid \underbrace{AR_m \leftarrow k.AS_k}_{\text{receive}}}^{\text{asynchronous}}$$

A formula in SY_m contains at least one occurrence of a synchronous action of an object belonging to the component denoted by m. Moreover, $SY_m \leftrightarrow k.SY_k$ expresses a synchronous calling of actions of objects from the distinct components denoted by m and k. AS_m and AR_m denote formulae containing at least one occurrence of a send and a receive communication action respectively. $AS_m \rightarrow k.AR_k$

states that m knows that the occurrence of a send action of an object in m implies that eventually there will be an occurrence of a corresponding receive action of an object in k. Conversely, $AR_m \leftarrow k.AS_k$ states that m knows that the occurrence of a receive action of an object of m means that sometime before a send action of an object of k occurred.

Component Descriptions

A component description $CD = (\Theta, Ax)$ is a pair containing a component signature Θ and a set of axioms in its corresponding component logic (i.e., $Ax \subseteq \text{MDTL}_\Theta$).

The Underlying Model

The semantics for the logic consists of labelled prime event structures: a truly concurrent model of computation (e.g., [13]). Each component has a model (a labelled prime event structure) associated with it where the events are labelled by formulae indicating the state of the component. We omit details on the model and semantics of MDTL for space reasons and since it is not essential for this paper. We refer the interested reader to [7].

5 FORMALIZING COMPONENTS

In this section, we show how component concepts are formalized using our approach. We use the introduced example for illustration.

Because component objects are essentially complex objects we need to consider that a component has associated with it a class with the same name. Consequently, component objects are actually instances of this class.

Interface specification

An interface specification can be seen as a component description where the component signature is a *view* component signature of another. An interface specification is given by $IS = (\Theta, Ax)$, where $\Theta = (\Sigma, \emptyset, \{(\Sigma, id)\})$ and Σ is a kernel signature with a unique component sort. The component sort corresponds to the interface type. The object sorts contain a sort for every class in the information model of the interface, and an additional sort for the interface type. There is only one constant component instance operation of the component sort and consequently only one component term. By contrast, there are several instances for the object sort associated to the interface (which denote component objects in the terminology of [2]). Attributes, associations and operations of the classes in the information model define corresponding object sort attribute and action operations. Interface operations are defined as operations for the class with the interface name. The axioms in the interface specification (Ax) denote pre and postconditions of the interface operations. Note that there are no imported signatures in an interface signature therefore

MDTL$_\Theta$ is given entirely by $m.H_m$ where m is the local component term (interface term). In other words, the axioms of an interface specification correspond to formulae in the home logic of the interface ($Ax \subseteq m.H_m$). A precondition is for instance written as the formula $m.(\triangleright o.a \Rightarrow o.att_1 = v)$ meaning that the operation a on an object o can only happen provided the object is in a state where its attribute att_1 has the value v. A postcondition on an operation a could be $m.(\odot o.a \Rightarrow o.att_2 = 100)$ and means that the value of attribute att_2 changes to 100 after the occurrence of the operation a on object o.

Recall the precondition of `whereParcel` of our example. Let $self$ be an instance of class ILocalizeParcel; ip and ic variables of data sort string; p an instance of class Parcel. The precondition in the interface specification ILocalizeParcel is given by the following formula in the corresponding home logic ($\forall$ quantifiers are omitted for simplification).

$$ILocalizeParcel.(\triangleright self.whereParcel(ip, ic, r) \Rightarrow \exists_p(p.id = ip \wedge p.customer = ic)))$$

Component specification

A component specification can be seen as a component description where the component signature contains as many export signatures as the offered interfaces and as many import signatures as the required interfaces. A component specification is given by $CS = (\Theta, Ax)$, where $\Theta = (\Sigma, Imp, Exp)$ and Σ is a kernel signature. The local component sort of Σ corresponds to the component specification type. The set of object sorts contains a sort for every class in the business type model and an additional sort for the component specification type. Attributes, associations and operations are treated as above. How the information models of the offered interfaces relate is given by kernel signature morphisms (in an export signature, an inclusion morphism describes how the symbols in the interface correspond to the symbols in the component specification signature). The constraints reflecting require dependencies and how operations are to be implemented are given by formulae in C_m where m is the local component term of the component specification. It can express communication with all the imported subcomponents.

Recall the two possible constraints imposed on the implementation of operation `whereParcel` in the component specification GIS. Let GIS denote the local component term of the specification, g an instance of class GIS; r a variable of data sort $Coordinates$; and q an instance of class IParcelInfo.

The synchronous operation calling in the context of the component specification GIS is given by a formula in the communication logic $GIS.C_{GIS}$.

$$GIS.(\odot g.whereParcel(ip, ic, r) \leftrightarrow IParcelInfo.(\odot q.where(ip, r)))$$

The asynchronous operation calling is given by the next formula in the same communication logic.

$$GIS.(\odot g.whereParcel(ip, ic, r) \rightarrow IParcelInfo.(\odot q.where(ip, r)))$$

Component Specification Architecture

A component specification architecture can also be given by a component description $CSA = (\Theta, Ax)$ where Θ is a component signature and all the components in the system correspond to an imported signature of Θ. Export signatures may exist if we have an open component architecture (as is the case in the ParcelCall example). The formulae in Ax describe: (1) the way components are connected, their dependencies and the way they interact with each other (through the communication logic), (2) internal component properties (through the home logic of the component).

6 CONCLUSIONS

In this paper we have shown how to uniformly formalize the specification of different component concepts like interfaces, components and component architectures. In particular, component contracts are described as formulae of the distributed logic MDTL: usage contracts are given by formulae in the home logic of an interface, whereas design contracts are captured by formulae in the communication logic of the component specification.

In this paper, we have restricted the presentation of our framework in several ways. We have not described the semantics underlying the logic because it is not essential for the purpose of the paper. Details can be found in [7]. Moreover, in our framework components can be parameterised by other components. This is omitted in the presentation as it is not covered by the approach in [2].

Several approaches in the literature provide a semantics to OCL including [10, 1, 3]. The latter is the only known logic-based approach. It uses BOTL, an object-based temporal logic, to formalise a subset of OCL. BOTL essentially results from combining CTL and an object logic in such a way that OCL expressions and constraints can be translated into it. MDTL compares to BOTL if we understand that the home logic of a module in MDTL is an object logic containing CTL. OCL does not, currently, allow to express more than static constraints on UML models. In particular, we have seen that OCL is not expressive enough to describe our component contracts as needed. Extending OCL to allow the description of such contracts would make BOTL insufficient as an underlying logical framework. By contrast, MDTL contains a communication logic which we believe to be as expressive as needed to describe interactions between components and/or objects. Whilst BOTL has been developed essentially for verification based on model checking, MDTL has not. MDTL is a logic for specification which is far to expressive to allow verification in general. For instance, MDTL contains past temporal operators as well as a concurrency operator which may be a problem for model checking (the so-called backtracking problem

which makes model checking undecidable). It is not clear that MDTL has indeed this problem, and this needs to be further investigated.

The ParcelCall system is an example of a complex open distributed system which when integrated with the legacy system of a transport or logistic company will contain dependability issues in the human-computer interactions. Notice that the carrier system includes humans and machines: the computer-based activities of the system involve entwined actions performed by several humans and machines. The localization system offered by ParcelCall will necessarily affect and influence the carrier's organisation and culture: working practices by human carriers will change, new dependability requirements (e.g., reliability, safety, security) on the system will emerge, etc.

Humans are normally part of the environment of a system. In dependable systems, however, it is crucial to integrate the human or the organisation (with several humans playing different roles and interacting with the system in different ways) in the specification. It is necessary to understand how the humans in the organisation behave and interact with the system in order to be able to assess or predict possible errors, faults, etc.

One approach that considers the human in systems where human-computer interactions are highly critical is the work by Rushby (as described e.g., in [11]). To try to analyse how errors can result from human-computer interaction, the approach compares what is called the *mental* model of an operator (system user) and the *system* model. The mental model corresponds to the model the operator believes to be the real model of the system. Both the system model and the mental model are described as finite state transition systems and checked for consistency using a mechanised formal method. The outcome of such a check suggests places where design should be improved.

In any case, to model human behaviour there is a need to borrow concepts and models from other disciplines like cognitive science and/or artificial intelligence. A combined formal approach can then be used to describe software systems, part of their environment, as well as their interactions. Verification tools based on such combined formalisms would make it possible to verify for instance quality of service and dependability constraints in systems with human-computer interaction.

An advantage of our component-based formalism is that it permits an integration of such aspects in a straightforward way. The human(s) can be regarded as a special kind of component with a distinct logic for describing her behaviour. The communication logic in MDTL as given can describe human-machine interactions in the same way. We are currently investigating an approach based on the distributed logic MDTL combined with agent logics (essentially logics of knowledge and belief) for describing relevant aspects of human behaviour. How feasible such an approach is for verification needs to be investigated.

REFERENCES

[1] M. Bidoit, R. Hennicker, F. Tort, and M. Wirsing. Correct realization of interface constraints with OCL. In R. France and B. Rumpe, editors, *The Unified Modeling Language — Beyond the Standard, Proc. 2nd Int. Conf., (UML'99) Fort Collins, CO, USA, Oct. 1999*, volume 1723 of *LNCS*, pages 399–415. Springer, 1999.

[2] J. Cheesman and J. Daniels. *UML Components*. Component Software Series. Addison-Wesley, 2001.

[3] D. Distefano, J.-P. Katoen, and A. Rensink. On a temporal logic for object-based systems. In S. F. Smith and C. L. Talcott, editors, *Formal Methods for Open Object-based Distributed Systems*, pages 305–326. Kluwer Academic Publishers, 2000.

[4] D. D'Souza and A. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Object Technology Series. Addison-Wesley, October 1998.

[5] J. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.

[6] A. Kleppe and J. Warmer. Extending OCL to include actions. In S. Kent and A. Evans, editors, *UML'2000 — The Unified Modeling Language: Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000*, volume 1939 of *LNCS*, pages 440–450. Springer, 2000.

[7] J. Küster Filipe. *Foundations of a Module Concept for Distributed Object Systems*. PhD thesis, Technical University of Braunschweig, Germany, September 2000.

[8] J. Küster Filipe. Fundamentals of a Module Logic for Distributed Object Systems. *Journal of Functional and Logic Programming*, 2000(3), March 2000.

[9] OMG Unified Modeling Language Revision Task Force. *OMG Unified Modeling Language Specification*, Version 1.4 draft, February 2001.

[10] M. Richters and M. Gogolla. OCL: Syntax, semantics and tools. In T. Clark and J. Warmer, editors, *Advances in Object Modelling with the OCL*, LNCS, pages 38–63. Springer, 2001.

[11] J. Rushby. Modeling the human in human factors. In *Proceedings of the 20th International Conference on Computer Safety, Reliability and Security (Safecomp 2001), Budapest, Hungary, September*, volume 2187 of *LNCS*, pages 86–91. Springer, 2001.

[12] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1997.

[13] G. Winskel and M. Nielsen. Models for Concurrency. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. 4, Semantic Modelling*, pages 1–148. Oxford Science Publications, 1995.

ABOUT THE AUTHORS

Juliana Küster Filipe is a research fellow at the University of Edinburgh, UK. She got her PhD from the Technical University of Braunschweig, Germany in 2000. She is currently working on two EPSRC funded projects on *Dependability of Computer-based Systems* and *Logic for UML*. She can be reached at jkf@dcs.ed.ac.uk. For more information about her research see http://www.dcs.ed.ac.uk/home/jkf.