

Plug-ins for ISpec

Practical and theoretical aspects of plug-in formalisms to express the requirements in the templates of the ISpec interface specification approach.

Louis van Gool, Hans Jonkers, Ruurd Kuiper, Erik Luit, Seguei Roubtsov

Abstract— ISpec is an interface specification approach where templates provide slots to write interface requirements. These requirements can be written in various "plug-in" formalisms. The practical question how to implement this in a tool is answered for regular expressions as a plug-in language. The requirements expressed by the regular expressions are used to assess the correctness of requirements expressed in sequence diagrams. In fact, an editor is coupled to the tool in which a plug-in language can be defined and a slot in a template can be linked to a particular language. The theoretical question how to formalise plug-ins in a relation calculus framework is investigated.

Keywords—Interface specification, component-based development, plug-in, formalization, CASE tool

I. INTRODUCTION

This contribution reports on theoretical and tooling investigations in component development carried out in PROGRESS project SpecTEC, in particular practical and theoretical aspects of plug-in languages. As the basis for our investigations we use the methodology for Interface Specification ISpec, developed at Philips Natlab [1]. Plug-ins are languages that can be plugged into ISpec specifications to express requirements of components at various levels of formality.

We briefly summarise some of the ideas of ISpec. An interface role diagram identifies the interfaces and the roles associated with them. The interfaces are introduced to provide communication between the roles. When we compare this to Object Oriented Modelling, the roles can be seen as the object classes and an interface role diagram can more or less be compared to a (UML) Class Diagram [2]. An interface can then be seen as a set of methods of one particular class and several interfaces will arrange the methods of a role (class) in any desired way. The roles can also have private methods, being methods not belonging to any of the interfaces of the role but rather methods of the role itself. These methods are not meant for external use, communication between the roles, but for internal use only.

When an interface "belongs" to a role we call it a Provided Interface of that particular role. The role provides the interface. On the other side of the communication there can be a roles "using" an interface. We call this a Required Interface of those roles. Furthermore we can have inher-

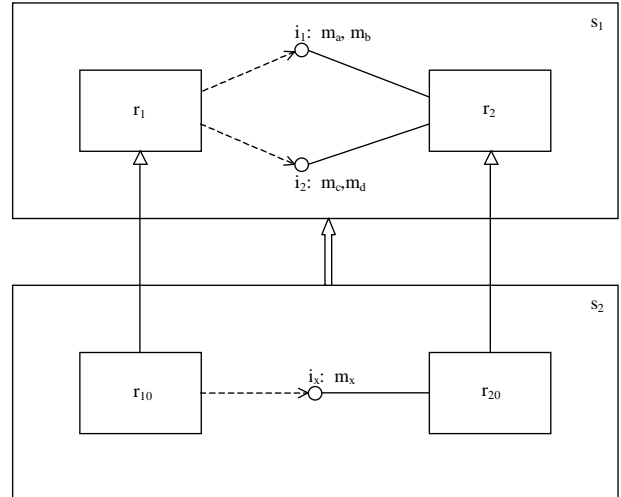


Fig. 1. An interface role diagram consisting of two interface suites

itance relations between roles for reusing or specialising purposes (figure 1), this follows the concept of inheritance relation we know from Object Oriented Modelling.

Templates are, essentially, documents with slots in which, for each of the elements present in the Interface-Role diagrams, a specification can be given. For example, the template for effect specifications of methods (figure 2) has slots for precondition, postcondition and action clause (describing the external call behavior); the role template contains, among others, a slot for invariants. Interface-Role diagrams plus the templates form the specification of an Interface suite. Many other formalisms, for example, graphical ones as in UML are used to provide various

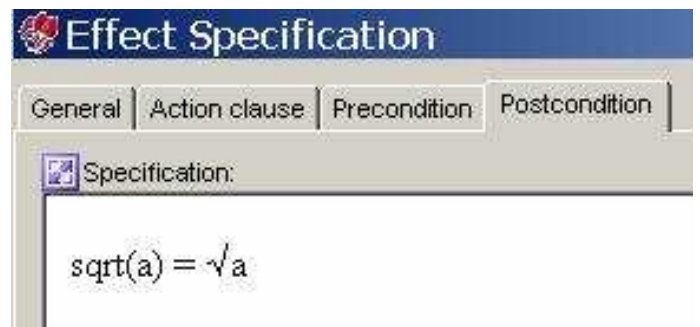


Fig. 2. Template for effect specifications

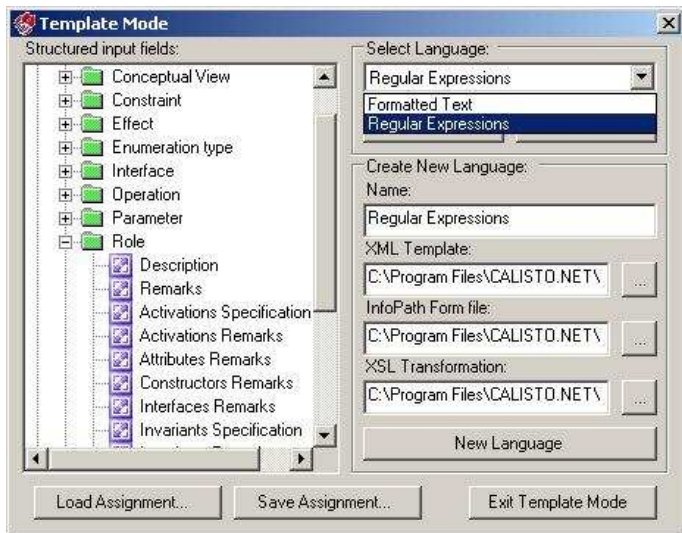


Fig. 3. Template Mode form - addition of regular expressions

views of the system, thus aiding the intuition. Consistency is with respect to the Interface-role diagram and the template information. Various formalisms to express the information in the slots can be chosen: plug-in languages.

In the SpecTEC project, a tool has been developed [3, 4] that supports various aspects of the ISpec approach. In the last year, the following activities regarding the tool can be reported. The first version was restructured; notably the event handling was generalized to allow easy extensibility. After this, the tool has been converted to Visual Basic .NET, it has been refactored into a full version (complete functionality) and a light version (only drawing functionality) and support for plug-ins, streams and Sequence Diagrams has been added [5]. Lastly, a pilot project was executed at Océ to assess ISpec and the tool in a different industrial environment [6]. Furthermore, the SpecTEC project provides theoretical underpinnings of the ISpec approach and tooling. The present document reports on the following two activities of the past year in more detail.

In section 2, the practical question how to implement plug-in languages in a tool is answered in a quite generic fashion, exemplified by regular expressions. In fact, an editor is coupled to the tool in which a plug-in language can be defined and a slot in a template can be linked to a particular language. The requirements expressed by the regular expressions will be used to assess the correctness of requirements expressed in sequence diagrams.

In section 3, the theoretical question how to formalise plug-ins in a relation calculus framework is answered.

Section 4 contains some conclusions.

II. TOOL SUPPORT FOR PLUG-INS

Plug-in languages are realized in the Calisto tool with the aid of InfoPath. InfoPath is a structured forms editor that is part of the Office 2003 suite. For a new language, an InfoPath form must be created and “published”. Publishing a form means that the form becomes available for use. In these forms, the syntax of the new plug-in language is defined. This means that every textual formalism can be added as a new language. The definition of the form and InfoPath’s functionality then ensure that only syntactically correct expressions can be entered. InfoPath generates an XML representation of the expression, which is returned to the Calisto tool. This expression can then be used for, e.g., consistency checks. For example, it can be verified that the role, interface and operation names used in expressions actually exist in the model. A more sophisticated check that we are currently implementing is the consistency of the Sequence Diagram view with respect to the regular expressions in action clauses.

The Calisto tool was extended with Template Mode, in which new plug-in languages can be added in the manner described above (figure 3). Template Mode also enables to couple slots in the templates to the different plug-in languages.

Calisto provides more user support than just a syntax check on the expression. It also makes model information available to InfoPath; so-called context information. To give an idea of this context information, consider figure 1. In this figure, two interface suites are shown. The lower suite inherits from the upper suite. The upper suite is referred to as a base document in the Calisto tool. The context information passed to InfoPath consists of the names of all roles, interfaces and streams and whether these are imported from a base document. For interfaces, the names of all operations and parameters are included as well.

When a slot is edited, Calisto writes the context information to an XML file, after which InfoPath is opened with

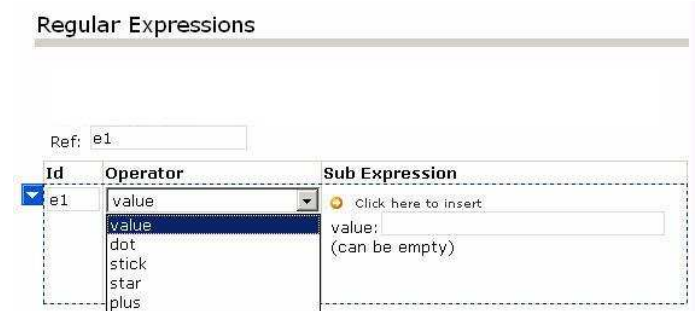


Fig. 4. InfoPath form for Regular Expressions

the form for the language coupled to the slot. After the expression is entered, InfoPath passes its XML representation to Calisto. In the current implementation of regular expressions (figure 4), the context information is not yet used. However, it is relatively simple to use the context information to, e.g., fill drop-down lists.

III. FORMAL TREATMENT OF PLUG-INS

The formal semantics we have developed for ISpec allows for a straightforward theoretical treatment of the concept of a plug-in. Our semantics for ISpec is *denotational*. This means that each *syntactical* construct in ISpec is directly associated with a *semantical* construct in our formalism. If we look for example at the definition of a ‘method’ in terms of a ‘guarantee’ and an ‘assumption’, this would be syntactically represented as

$$\text{method}(grnt, assm)$$

where $\text{method}(_, _)$ is a construct that takes two other syntactical constructs (indicated by the variables $grnt$ and $assm$) and delivers a new syntactical construct $\text{method}(grnt, assm)$.

In a denotational semantics, a *semantics function*, usually denoted by $\llbracket _ \rrbracket$, takes a syntactical construct and maps it onto a semantical construct, recursively applying itself to the arguments of the syntactical construct. The semantics of $\text{method}(grnt, assm)$ would then for example be defined by

$$\begin{aligned} & \llbracket \text{method}(grnt, assm) \rrbracket \\ = & \\ & \llbracket grnt \rrbracket / \llbracket assm \rrbracket \end{aligned}$$

where $/$ is some semantical operator that defines how the semantics of a ‘guarantee’ and the semantics of an ‘assumption’ should be combined.

This approach to denotational semantics is actually not so smart. As you can see in the above example, we now have two representations of a method. A syntactical one, being

$$\text{method}(grnt, assm)$$

and a semantical one, being

$$\llbracket post \rrbracket / \llbracket pre \rrbracket$$

the last one being equal to

$$\llbracket \text{method}(grnt, assm) \rrbracket$$

If we would simply only use the semantical representation, we do not have to do things twice and can get rid of the annoying semantics function $\llbracket _ \rrbracket$. For the above example, this means that we define a *semantical* function method that takes two *semantical* arguments $grnt$ and $assm$. The definition of this function is then simply the semantics of our ‘syntactical’ construct:

$$\begin{aligned} & \text{method}.\langle grnt, assm \rangle \\ = & \\ & grnt / assm \end{aligned}$$

Now suppose that we have a function guarantee that constructs a semantical ‘guarantee’ from an *action clause*, a *postcondition* and a *result type* and a function assumption that constructs a semantical ‘assumption’ from a *precondition* and *parameter types*, defined by

$$\begin{aligned} & \text{guarantee}.\langle action, post, resultType \rangle \\ = & \\ & action \cap post \cap resultType \end{aligned}$$

and

$$\begin{aligned} & \text{assumption}.\langle pre, paramsType \rangle \\ = & \\ & pre \cap paramsType \end{aligned}$$

The functions *method*, *guarantee* and *assumption* can now be used to combine an action clause, a postcondition, a result type, a precondition and parameter types into a single method as follows:

$$\begin{aligned} & \text{method}.\langle \text{guarantee}.\langle action, post, resultType \rangle \\ & \quad , \text{assumption}.\langle pre, paramsType \rangle \\ & \quad \rangle \end{aligned}$$

As you can see, combining these semantical functions is actually no different from combining syntactical constructs. The advantage is however that we do not have to define things twice and we do not have the annoying $\llbracket _ \rrbracket$. A formula that could be called the ‘semantics’ of the above formula can be obtained by simply writing out the definitions of the constituent components:

```

method.<guarantee.<action, post, resultType>
    , assumption.<pre, paramsType>
    >
=    {definition of guarantee and assumption}
method.<action  $\cap$  post  $\cap$  resultType
    , pre  $\cap$  paramsType
    >
=    {definition of method}
    (action  $\cap$  post  $\cap$  resultType) /
    (pre  $\cap$  paramsType)
    
```

We wrote ‘semantics’ between ‘and’ because the equal signs = show that the formula already ‘is’ its semantics.

This shows our approach to the description of a formal semantics for ISpec. We define a set of functions that we call *combinators*, each of which corresponds to a construct in ISpec. Now how can we represent the concept of a plug-in? The answer to this question is not very difficult. A plug-in simply is some function that provides the input of some combinator. If we would have for example some function *exp* that transforms a string representing a boolean expression, into a function that accepts some *context* in which the (boolean) value of the string is determined, then this function *exp* enables us to ‘plug-in’ strings that represent for example a precondition. As an example, we could have

```

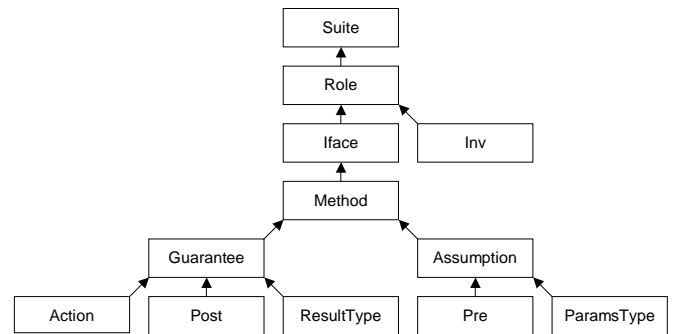
method      = method.<grnt, assm>
grnt       = guarantee.<action, post, resultType>
assm      = assumption.<pre, paramsType>
action     = ...
post      = exp.("(<self -> new a) =
                (<self -> old a) - 1")
resultType = ...
pre       = exp.("(<self -> a) > 0")
paramsType = ...
    
```

As told, the function *exp* transforms an expression *E* into a function *exp.E* that accepts a certain context *c* and delivers some boolean value *b*. The context is the thing that enables the evaluation of the expression. It should contain the value of “self” (representing the “self object”) and the value of a state that enables one to determine the value of “attribute” *a* of this self object (or in case of the post-condition, the new and old value of this attribute). For the precondition, this context would look like (g, o) , *g* being a ‘global state’ containing a collection of objects and *o* being the current ‘self object’. For the ensures clause, the context would look like $((g, h), o)$ where *g* is the ‘new global

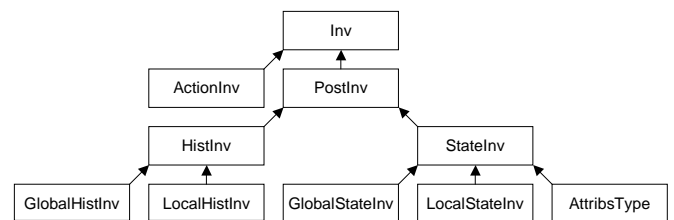
state’, *h* the ‘old global state’ and *o* again the current ‘self object’.

Now we have explained how this principle of combining combinators works, it is time to show that this approach also works for an industrial language like the ISpec language. The biggest challenge was to develop appropriate *semantical domains* for the combinators that describe the ISpec language, like the (g, o) and $((g, h), o)$ of the above example. Not only do we want to give a semantics for ISpec, but the semantics should of course be in line with the intuition of the founder of this language (Hans Jonkers, with whom we had regular meetings to check our understanding of the language). Furthermore, the language should have nice algebraic properties, because a language without nice algebraic properties is doomed to be misinterpreted by the users of it.

Before we show the definitions of the combinators for ISpec, we first show two pictures that show which combinators provide the input of which other combinators. The first picture shows how a ‘suite’ is constructed:



The names in this picture represent the input/output types of the combinators. For example, the inputs of the combinator that constructs a thing of type *Role* are of type *Iface* and *Inv*. The types that have to do with ‘invariants’ are described in the next picture:



We do not delve deep into our theory to explain every detail of all these types and combinators. All this can be read in the thesis [7]. We give however the types of the combinators and one definition here, just to give a feeling of the complexity of these combinators. Actually, reducing this complexity to a minimum is what most of the thesis is all about. However, a certain amount of complexity is needed to give the language the properties it should have.

One of the main types that plays a role in all this, is the type *State*. This type describes what the state of a specification (which can be seen as a high-level program) incorporates. We decided it to consist of actually only four elements (g, o, y, w) . Element g represents the *global state*, describing the *local state* of all objects in the system. Element o identifies the *self object*, the object that is currently active. Element y represents the result value of a method and element w represents the value of the parameters. The types in the pictures are now defined by

| | |
|----------------|---|
| Suite | = $S(((\text{State}) \rightarrow (\mathbb{M})) \rightarrow (\mathbb{I})) \rightarrow (\mathbb{R}))$ |
| Role | = $S((\text{State}) \rightarrow (\mathbb{M})) \rightarrow (\mathbb{I})$ |
| Iface | = $S(\text{State}) \rightarrow (\mathbb{M})$ |
| Inv | = $S(\text{State})$ |
| ActionInv | = $S(\text{State})$ |
| PostInv | = $P(\text{State} \star \text{State})$ |
| HistInv | = $P(\text{State} \star \text{State})$ |
| GlobalHistInv | = $P(\text{State} \star \text{State})$ |
| LocalHistInv | = $P(\text{State} \star \text{State})$ |
| StateInv | = $\wp(\text{State})$ |
| GlobalStateInv | = $\wp(\text{State})$ |
| LocalStateInv | = $\wp(\text{State})$ |
| AttribsType | = $\wp(\text{State})$ |
| Method | = $S(\text{State})$ |
| Guarantee | = $S(\text{State})$ |
| Assumption | = $\wp(\text{State})$ |
| Action | = $S(\text{State})$ |
| Post | = $P(\text{State} \star \text{State})$ |
| ResultType | = $\wp(\text{State})$ |
| Pre | = $\wp(\text{State})$ |
| ParamsType | = $\wp(\text{State})$ |

Again, we will not delve deep into the meaning of the used symbols, but only give a short informal description. The bold letters \mathbb{R} , \mathbb{I} and \mathbb{M} represent the types of the names of roles, interfaces and methods respectively. The definition of *Iface* roughly means that the things that an interface consists of (what these things are, is defined by the combinator that constructs an interface; these things will be methods of course), are labeled by method names. Similar for *Role* and *Suite*. The S depicts the fact that within suites, roles, interfaces, (action) invariants, methods, guarantees and action clauses, we need to be able to call other methods within the same suite. The thesis defines exactly how this is formalised by means of fixed-point theory. Then we have the type $P(\text{State} \star \text{State})$ that represents something that talks about two states (a state change) and things of type $\wp(\text{State})$ that talk about a single state.

This should be enough for a vague understanding of

what is going on. A bit more concrete description is provided by the next table that describes the types of the combinators:

| | |
|-------------|--|
| suite | $\in \text{Suite} \leftarrow (\text{Role}) \rightarrow (\mathbb{R})$ |
| role | $\in \text{Role} \leftarrow (\text{Iface}) \rightarrow (\mathbb{I}) \times \text{Inv}$ |
| iface | $\in \text{Iface} \leftarrow (\text{Method}) \rightarrow (\mathbb{M})$ |
| inv | $\in \text{Inv} \leftarrow \text{ActionInv} \times \text{PostInv}$ |
| actionInv | $\in \text{ActionInv} \leftarrow S(\text{State})$ |
| postInv | $\in \text{PostInv} \leftarrow \text{HistInv} \times \text{StateInv}$ |
| histInv | $\in \text{HistInv} \leftarrow \text{GlobalHistInv} \times \text{LocalHistInv}$ |
| stateInv | $\in \text{StateInv} \leftarrow \text{GlobalStateInv} \times \text{LocalStateInv} \times \text{AttribsType}$ |
| attribsType | $\in \text{AttribsType} \leftarrow (\wp \mathbb{V}) \rightarrow (\mathbb{A})$ |
| method | $\in \text{Method} \leftarrow \text{Guarantee} \times \text{Assumption}$ |
| guarantee | $\in \text{Guarantee} \leftarrow \text{Action} \times \text{Post} \times \text{ResultType}$ |
| assumption | $\in \text{Assumption} \leftarrow \text{Pre} \times \text{ParamsType}$ |
| resultType | $\in \text{ResultType} \leftarrow \wp \mathbb{V}$ |
| paramsType | $\in \text{ParamsType} \leftarrow (\wp \mathbb{V}) \rightarrow (\mathbb{Q})$ |

You might notice that for the types *GlobalHistInv*, *LocalHistInv*, *GlobalStateInv*, *LocalStateInv*, *Action*, *Post* and *Pre* there is no combinator in this table. These are actually the places that are considered to be ‘plug-in’ places. These places are not in any way forced by our formalism, but represent a choice where we define the border between *ISpec* and plug-in language. The mathematical language that is developed in the thesis and that is also used to describe the meaning of the combinators for *ISpec* provide powerful constructs to create these kind of plug-in languages. In one thesis chapter we define for example a framework that can incorporate many different kinds of denotational expression languages, especially focussing on flexibility and powerful ways to deal with partiality and even non-determinism. There are also many relation-algebraic constructs that can be used to formally define the meaning of the regular expressions that have been implemented in the tool.

We will not give the formal definition of all combinators for *ISpec*, but take out one that illustrates some interesting aspects of the formalism. The combinator *role* for constructing roles, is formally defined by

$$\begin{aligned} & \text{role}.\langle \text{ifaces}, \text{inv} \rangle \\ & = \\ & \hat{\Pi}_g \text{ifaces} \hat{\Pi} (\text{inv} \hat{\Leftarrow}_g \mathbb{I}) \hat{\Leftarrow}_g \mathbb{M} \end{aligned}$$

Without knowing the meaning of $\hat{\Pi}_g$, $\hat{\Pi}$ and $\hat{\Leftarrow}_g$, this formula does not say much of course, so we give an informal explanation of these operators.

We start with the operator $\hat{\Pi}_g$. As can be seen from the table that gives the types of the combinators, the type of

ifaces is $(\text{Iface} \multimap \mathbb{I})$. This roughly means that *ifaces* is a collection that consists of (interface behaviour, interface name)-pairs. The operator $\hat{\cap}_g$ transforms this collection *ifaces* into the mathematical object $\hat{\cap}_g \text{ifaces}$ that has nicer algebraic properties than this collection, but still represents this collection.

Now for the operator $\hat{\cap}$. This operator can sort of be read as “and”. The meaning in the formula $\hat{\cap}_g \text{ifaces} \hat{\cap} (\text{inv} \hat{\triangleleft}_g \mathbb{I}) \hat{\triangleleft}_g \mathbb{M}$ roughly is that next to the behaviour of the interfaces, also the invariants should hold.

Finally the operator $\hat{\triangleleft}_g$ makes it possible to distribute behaviour over a collection of things. The expression $(\text{inv} \hat{\triangleleft}_g \mathbb{I}) \hat{\triangleleft}_g \mathbb{M}$ distributes the invariant for example over all methods of all interfaces of the role in question. The operator $\hat{\triangleleft}_g$ is again of course defined in such a way that it satisfies nice algebraic properties.

Next to the meaning of the operators, another question that comes quickly to mind is why all these embellishments like $\hat{\cap}$ and $\hat{\triangleleft}_g$ are there. First of all, the three operators $\hat{\cap}$, \cap and \triangleleft also exist in our formalism, but the semantical domain in which they live is simpler than the one needed for the semantics of ISpec. Although the operators $\hat{\cap}_g$, $\hat{\cap}$ and $\hat{\triangleleft}_g$ are very similar to \cap , \cap and \triangleleft respectively, their definitions are of course a bit different because they have other semantical domains. In the thesis, we already used a quite advanced type system to keep these embellishments to a minimum, but a few can never be avoided (unless we use overloading of course, but that is considered a bad habit).

This ends our formal treatment of ISpec. More detailed information can be found in the thesis.

IV. CONCLUSIONS

Various diagram languages are used to describe interface requirements. Templates combine this information into one formalism, thus enabling to establish or maintain (e.g during design) consistency of the various descriptions. The practical result is, that for sequence diagrams versus regular expressions this is achieved, the theoretical result provides a model to prove correctness of the tool. In general, this is useful to achieve consistency in specifications; in particular, tool and methodology are aimed to be deployed at Philips.

REFERENCES

- [1] H.B.M. Jonkers, Interface-Centric Architecture Descriptions, In proceedings of WICSA, The Working IEEE/IFIP Conference on Software Architecture (2001), pp. 113-124.
- [2] P. Stevens and R Pooley. Using UML. Software Engineering with Objects and Components. Addison-Wesley, an imprint of Pearson Education.

- [3] K. van Gogh, R. Kuiper and E.J. Luit, Consistency in ISpec Specifications, Proceedings of the 4th Progress Symposium on Embedded Systems, October 22, 2003.
- [4] Calisto documentation, <http://www.win.tue.nl/calisto>
- [5] Arkas documentation, <http://www.win.tue.nl/calisto>
- [6] F. Kratz, Grizzly i-spec, Technische Universiteit Eindhoven, 2004.
- [7] L.C.M. van Gool, Formalisation of practical specification concepts, phd-thesis, Technische Universiteit Eindhoven (draft).