A Classification of Pointcut Language Constructs

Maximilian Stoerzer
University of Passau
Passau, Germany
stoerzer@fmi.uni-passau.de

Stefan Hanenberg
University of Duisburg-Essen
Essen, Germany
shanenbe@cs.uni-essen.de

ABSTRACT
Aspect-oriented systems provide pointcut languages in order to specify selection criteria for join points which in turn will be adapted. However, a closer look into current pointcut languages reveals that there are large differences among them. Consequently different aspect-oriented system permit to specify different selection criteria. This also means that it is in general hard to state whether a certain aspect-oriented system is adequate for a given problem without detailed system knowledge.

This paper analyzes and classifies pointcut language constructs based on the objects they reason on. Based on this analysis, we propose three conceptual classes of pointcut constructs. These classes represent an abstract framework for pointcut languages allowing to better understand and compare existing approaches. They also describe a design space for potential new language constructs.

1. MOTIVATION
Aspect Oriented Programming (AOP) as first introduced in [13] addresses the problem of crosscutting concerns. The term crosscutting concern describes parts of a software system that logically belong to one single module, but which cannot be modularized due to limited abstractions of the underlying programming language. Aspect-oriented software aims to overcome the problem of crosscutting concerns by introducing a new kind of module - the aspect.

Aspects extend the underlying application by providing additional functionality “at certain points”. These points are called join points in the aspect-oriented terminology. In order to specify where aspects extend the base application, aspect-oriented systems provide language constructs that permit to select those join points where aspects should be woven to. In correspondence to [12, 5, 11] the term pointcut language is used to describe these selection languages. In order to specify how a certain selected join point should be adapted, aspect-oriented systems provide additional language constructs like advice in AspectJ.

Meanwhile, there are a number of systems available permitting to develop software in an aspect-oriented way, like AspectJ [13, 12], Hyper/J [9, 4], AspectS [11] or Sally [8]. Although all of them provide aspect-oriented features – the selection and adaption of join points – there are a number of differences among them.

First, different aspect-oriented systems provide different kinds of join points. For example, AspectJ permits to select those points in the execution of a program where an object’s field value is set. Approaches like for example Hyper/J or AspectS do not permit to select these kinds of join points.

Second, the features distinctive pointcut languages in current approaches provide to select join points differ. For example, the cflow-construct in AspectJ (allowing to select join points based on properties of the call-stack) is a feature that is not directly available in Hyper/J or Sally.

Third, different aspect-oriented systems differ in the kind of adaptations they provide for each kind of join point. For example, AspectJ provides the proceed-construct within around-advice which permits to decide at runtime whether or not execution should proceed with the original join point. A similar join point adaptation does not exist for example in Hyper/J.

These different facets of aspect-oriented systems make it hard to compare them. As a consequence, it is hardly possible to determine whether or not a certain aspect can easily be implemented in a given system. Furthermore, whenever a new proposal for a language constructs appears, it is hard to determine whether this feature differs conceptually from known ones. The overall problem is that conceptual models are missing that permit to compare different aspect-oriented systems.

In this paper we put the focus the different pointcut language constructs and abstract as far as possible from the underlying join point model and the adaption mechanism. We propose a classification of pointcut language constructs (pointcut constructs for short) which provides a conceptual view on them. These classes also permit to classify aspect-oriented systems based on the features provided by their pointcut languages. Furthermore, they represent an abstract, general framework for the development of pointcut languages.

In section 2 we briefly discuss different facets of aspect-oriented systems and introduce a simple execution model our classification uses. In section 3 we introduce our classification of pointcut constructs. Section 4 applies the classification to a number of aspect-oriented systems - namely AspectJ, Hyper/J, Sally and AspectS. After referring to re-
2. SETTING THE SCENE

Before we describe the different construct classes we first introduce a general model of aspect-oriented systems where we describe the different ingredients an aspect-oriented system consists of. The later on proposed classification is closely related to the terms of static and dynamic join points. Hence, this section also briefly introduces these terms. Furthermore, we describe a general program execution model that is used when referring to dynamic program behavior.

2.1 A Conceptual Model for Aspect-Orientated Systems

In order to study different aspect-oriented systems in respect to their expressiveness it is necessary to have a conceptual understanding of aspect-oriented systems. As a base for our discussion, we developed an abstract model of aspect-oriented systems by factoring out three core components:

Join Point Model: The join point model defines the join points available for adaption in a specific system.

Pointcut Language: The pointcut language is the query language to select a subset of the join points defined by the join point model.

Adaption Mechanism: The adaption mechanism provides means to add or modify functionality at selected join points.

The term join point model includes all elements of an aspect-oriented system that can be selected and adapted. Elements of the join point model are for example method definitions in Hyper/J or method calls in AspectJ. Examples for pointcut languages are the pointcut language of AspectJ or the constructs within a concern mapping and hypermodule in Hyper/J. Adaptation mechanisms are advice or inter type declarations in AspectJ, or composition rules in Hyper/J.

2.2 Static versus Dynamic Join Points

The term join point is defined in [12] as a “principled point in the execution of a program”. However, in the aspect-oriented literature there is already the notion of static and dynamic join points [2]. A static join point can be characterized as a location in the program’s source code (15) describes join points as “systematic loci” of a program, [1] describes join points as “elements of a program”).

The characteristic of static join points is that the selection (and adaption) of a certain element only depends on selection criteria referring to the application’s static structure. For a given syntactic element and a specific pointcut expression it can be unambiguously determined whether or not it should be selected (and adapted). I.e. every time the corresponding source code elements is reached/executed at runtime the same adaptation is performed.

This differs for dynamic join points. These join points do not correspond directly to elements in the application source, but may have an associated source code element. In [15] this element of application’s sources has been called a join point shadow. However, we will not use this term here as

a “shadow” itself can be a valid static join point (depending on the join point model).

While static join points address the locations in source code available for an aspect, each static join point can be reached a multitude of times during program execution. We define a dynamic join point as a single hit of a static join point during program execution. One gets a good impression of this idea by thinking of a program trace, which records each static join point every time it is reached during program execution.

The characteristic feature for dynamic join points is that conditions that need to be evaluated at runtime and that check whether or not the join point should be adapted are implicitly expressed in the pointcut language. In [7] we refer to such runtime conditions as join point checks. In [10] the term “dynamic residue” has been used.

To summarize, the difference between static and dynamic join points is that the decision whether or not a join point should be adapted depends on runtime information. Consequently, a system that provides dynamic join points needs to provide a different kind of pointcut language, as this language needs constructs to refer to runtime values. In contrast to that, a system that provides static join points needs to provide means to reason on an abstraction of the source code.

2.3 Modeling Dynamicity: a Program Execution Model

Dynamic join points access the system state. However, ‘system state’ is a rather fuzzy term which has to be discussed. Therefore we introduce a simple model describing program execution as a sequence of system states. Each state σ is associated with an environment envσ providing a mapping from names known in the system (all declared variables/functions/etc.) to values, and a statement s that will be executed next. We refer to the set of known names with names(envσ) For v ∈ names(envσ) : val(v, σ) allows to access the value for a given name for a state σ.

The starting state is σ0, where only the runtime environment has been initialized but no user code has been executed yet. Thus the environment envσ0 contains no values (each name is associated with an initial/null value) and the corresponding statement s represent the first statements in the program.

The evaluation of a program statement s results in a state transition

σi ̸−−−→ σi+1

and might potentially change variable values in envσi, resulting in a new environment envσi+1. The execution of a program for a given set of input values thus results in a state sequence

σ0 ̸−−−→ σ1 ̸−−−→ σ2 ̸−−−→ . . . ̸−−−→ σn

where σn is the final state where the program terminates.

Note that due to loops and recursion s1 = sj for j ̸= i is
Although the concrete semantics of a statement \( s \) are defined by the semantics of the underlying base language and left open here, we can state the effects of a statement \( s \) more precisely by associating a context with each statement \( s \).

**Definition 2.1 (Statement Context).** The context of a statement \( s \) is defined as:

\[
\text{context}(s) \subseteq \text{names}(\text{env})
\]

Note that context \( (s) \) depends on the concrete semantics of \( s \). For a Java method call, the context would include all formal parameters (including the target object), the this-reference, all local variables within the current scope and all global variables. This also includes a syntactic representation of the stack-trace that can be generated in Java by Exception instances. In Smalltalk the available context is much larger: Smalltalk provides the special variable thisContext that permits to access objects that occur in the current control flow (in contrast to the pure syntactical call-stack representation available in Java).

Note also that the context of a statement \( s \) is statically defined – the set of accessible names does not change during system execution, thus context \( (s) \) does not depend on the current system state. However the set of associated values changes, and consequently depends on the system state. We use this observation now to state the effect of a statement \( s \) on the environment.

**Definition 2.2 (Effect of a Statement).** A statement \( s \) potentially modifies values in its context:

\[
\sigma_i; \text{env}_{s_{i+1}}; s; \text{context}(s) \subseteq \text{names}(\text{env}_{s})
\]

\[
\sigma_{i+1}; \text{env}_{s_{i+1}} = \text{env}_{s_i}[\text{val}(v, \sigma_i)\langle s(\text{val}(v, \sigma_i)) \rangle] : \forall v \in \text{context}(s)
\]

In the above definition we model the semantics of \( s \) as a function defined on the environment, which can change values for names accessible through context \( (s) \): by executing a statement the values of variables in the environment are substituted by the values that are part of the statement’s context. For values not touched by \( s \), we assume that \( \text{val}(v, \sigma_{i+1}) = \text{val}(v, \sigma_i) \).

### 2.4 Join Points and Join Point Model

The intention of aspect-oriented systems with state-based pointcut constructs is to specify selection criteria that depend on the system state \( \sigma \), or rather the associated data in the environment \( \text{env}_{\sigma} \): this implies that they rely on dynamic join points. We define a dynamic join point in the previously described model as a tuple \( (s, \sigma) \).

**Definition 2.3 (Dynamic Join Point).** A dynamic join point \( jp \) is defined as:

\[
jp = (s, \sigma)
\]

The statement \( s \) represent the statement that will be executed next, and \( \sigma \) represents the state. This corresponds to the definition of join points as principled points in the execution of a program [12]. The different join point models underlying aspect-oriented system provide different kinds of join points. For example, in AspectJ assignments to a local variable do not represent join points, while field assignments do. In AspectS, field assignments do not represent join points at all. The kinds of join points naturally depend on the underlying join point model and the base language.

As a consequence, not each tuple \( (s_i, \sigma_i) \) in the state sequence \( \sigma_0 \overset{n}{\rightarrow} \sigma_1 \overset{1}{\rightarrow} \sigma_2 \overset{2}{\rightarrow} \ldots n-1 \overset{n}{\rightarrow} \sigma_n \) necessarily forms a dynamic join point. This depends on the underlying kind of statement in conjunction with the system’s join point model.

### 3. A Classification of Pointcut Constructs

Systems based on static join point models like Hyper/J or Sally permit to select join points only because of characteristics that can be checked at compile-time. At the basis of the design document but rather represents any abstraction of the source code. Method calls are selected and adapted because of their position in the code, classes are selected and adapted because of their name (Hyper/J) or because their relationship to other classes (Sally).

Systems like AspectJ or AspectS provide the ability to specify runtime selection criteria. However, such selection criteria also have different qualities. For example, a selection in AspectJ based on the pointcut designators this, target and args only permits to specify the actual runtime types of objects participating in a certain join point. By using the if-pointcut designator instead it is possible to specify a selection criterion that refers to the arbitrary runtime values of the system.

However, the pointcut designator cflow (which also exists in AspectS) is a different kind of join point selection: the selection criterion does not only depend on the system’s current state in terms of the objects participating in the join point, but on the call-stack of the abstract machine executing the program when the join point is reached. The call-stack itself however represents a part of the history of a program.

Conceptually, the cflow construct differs from constructs like if, this, target and args, because it refers to passed states in the execution history of a program. Extensions of AspectJ like for example the dataflow-pointcut designator as described in [14] are comparable: again the selection criterion reasons about the execution history of the program.

Based on these observations, we define different classes of pointcut constructs that differ in the quality of the selection criteria they permit to express:

**Specification-based.** A specification-based pointcut construct permits to specify criteria for join points that refer only to the specification of an application.

**State-based.** A state-based pointcut construct permits to specify criteria for join points that refer to the program’s current state (i.e. runtime values).

**Progress-based.** A Progress-based pointcut construct permits to specify criteria for join points that refer to the progress of an applications execution.

In the subsequent sections these classes of pointcut constructs will be discussed in detail.

---

\(^4\)Here, the term specification does not necessarily refer to a design document but rather represents any abstraction of the source code.
3.1 Specification-based Join Point Languages

Specification-based selection constructs form the most simple and best understood form of pointcut constructs. Such systems have a static join point model and join points can be selected based on a static view on the underlying system. Systems that are based on such languages provide abstractions of the source code to be selected and adapted. Consequently, systems based on specification-based join point selections do not refer to any runtime information of the underlying base system.

Definition 3.1 (Specification-based construct). Specification-based pointcut constructs allow to select join points based on an abstraction of the source code.

Examples for specification-based constructs are the features provided by the pointcut languages of Hyper/J or Sally, since both only refer to elements in the application’s sources from within their selection language. In Sally classes can be selected because of their occurrence as formal parameters in method definitions, or because of their occurrence as types of fields within different class definitions. In contrast to this, Hyper/J selects classes because of their names, or because of their super-classes.

Note that here not necessarily the whole source code (in terms of the syntax tree) has to be available. Actually there is a wide range of abstraction levels for specification based constructs. For example it is possible that only class names or relationships between classes (like inheritance relationships) are available as base information to select join points with. Program rewriting systems6 on the other hand often allow a more detailed view on the source code.

Hence, aspect-oriented systems based on specification-based pointcut constructs can widely differ with respect to their expressiveness. However, as a commonality, all these systems select join points based on some source code abstraction only.

3.2 State-based Constructs

State-based constructs in general refer to a dynamic program execution model and consequently also dynamic join points. When examining these approaches in detail, it is observable that different aspect-oriented systems provide different data from the environment that state-based constructs can refer to. For example, a method call in AspectJ can be selected because of criteria specified for the caller object, the called objects and the formal parameters. But it is for example not possible to refer to the local variables within the current scope of a statement s. Consequently, an aspect-oriented system provides a context for join points that does not necessarily correspond to the statement’s context.

Definition 3.2 (Join Point Context). A join point context for a given join point \(j_p\) is a subset of context defined by a statement \(s\):

\[\text{context}(j_p) \subseteq \text{context}(s)\]

Hence, a join point’s context is a subset of the context defined by the underlying statement. Based on this definition, we define state-based pointcut constructs as follows.

Definition 3.3 (State-based Constructs). State-based pointcut constructs allow to specify selection criteria for a dynamic join point \(j_p = (s, \sigma)\) based on variable values defined in the join point’s context:

\[\text{val}(v, \sigma), v \in \text{context}(j_p)\]

The characteristic element of state-based constructs is that they permit to specify selection criteria for a join point based on the system’s state at the corresponding join point (hence we use the term current state). In general, different pointcut languages of different aspect-oriented systems can be classified according to the size and quality of the context for a join point. Obviously, the more information is available in the context, the more fine-grained selection criteria can be specified. In general, the number of variables can vary between one to the number of variables defined by the environment. In the latter case however the concept of join point context is superfluous.

Besides the context size it is observable that different aspect-oriented systems based on state-based pointcut constructs differ in respect to the quality of context values accessible by the pointcut language. For example, earlier versions of AspectJ (that did not contain the if-pointcut) permitted to specify selection criteria only for the runtime types of objects that are accessible via the join point context (using the above mentioned pointcut designators). I.e. it was not possible to specify for example field values of participating objects as selection criteria. Current AspectJ versions also offer access to values using the if pointcut designator.

3.3 Progress-based Pointcut Constructs

Specification-based and state-based pointcut constructs already permit to classify the features provided by a wide range of aspect-oriented systems. However, it is observable that a certain kind of pointcut designators does not fit into these categories. For example, the proposed dataflow-pointcut14 not only refers to the current state of the system, but also to a state that once was reached. I.e. the values the pointcut construct refers to in order to select a join point \(j_p\) do not necessarily come from the environment \(env_{s_i}\), but from an environment \(env_{s_j}\) (where not necessarily \(j = i\)).

From our point of view this represents a different class of pointcut construct that we call progress-based pointcut constructs.

6It is theoretically possible that a system provides a larger context than defined by the underlying statement, like for example adding additional variables from the environment. However, until now there is no system known to us that provides such functionality. Additionally, allowing to access the whole environment would somewhat invalidate the concept of a dynamic join point, as you could basically always monitor the state completely independent from any statement.

7In case the number of variables is zero, the pointcut language is not able to specify any runtime-specific condition on the join point. Consequently, all join points (of a certain kind) would be selected; this corresponds to a static join point selection.

8I.e. the maximum number of variables is the number of variables defined in the environment (whereby the definition of what variables belong to the environment depends on the semantics of the underlying language).
4.4 AspectJ

AspectJ currently is the most popular aspect-oriented system based on the programming language Java. Interestingly, AspectJ provides a pointcut language that contains elements from a hyperslice by their names in order to compose them using predefined composition rules like merge, override or bracket.

The elements Hyper/J selects in order to composed them represent pure static join points. Consequently, the pointcut language of Hyper/J is a pure specification-based pointcut language.

4.3 AspectS

AspectS is an aspect-oriented system based on the Smalltalk dialect Squeak. AspectS does not directly specify new language constructs for pointcuts, but a number of classes that can be used to specify join point selections.

A pointcut in AspectS consists of a so-called join point descriptor, which contains a class- and a method description representing the (static) join point (called shadow), and an advice-activator which represent the join point checks which execute whenever a join point shadow is reached.

The join point descriptor selects methods due to their static properties. Hence, it can be regarded as a language construct that belongs to the class of specification-based pointcut languages. The advice-activators on the other hand are executed at runtime and decide, based on the corresponding evaluation, whether the advice should be executed. Hence, advice activators belong in general the class of state-based pointcut languages.

Similar to AspectJ, AspectS provides a special advice activator to select join point because of selection criteria applied to objects on the call stack. However, Smalltalk already provides access to the call stack via a special variable. Consequently, this control-flow specific join point selection is – in contrast to AspectJ – a state-based construct, and no progress-based criterion. Because of that, AspectS can be regarded as an aspect-oriented system that provides a specification-based, as well as a state-based pointcut language.

4.2 Sally

Sally is a Java-based aspect-oriented system that provides a pure static join points. Consequently, the pointcut language of Hyper/J is a pure specification-based pointcut language.

4.1 Hyper/J

Hyper/J provides a pointcut language within its concern mapping, as well as in its hypermodules. Within the concern mapping file packages, classes, operations and fields are enumerated in order to be added to a hyperslice. Within the hypermodule specification it is possible to refer to the elements from a hyperslice by their names in order to compose them using predefined composition rules like merge, override or bracket.

The elements Hyper/J selects in order to composed them represent pure static join points. Consequently, the pointcut language of Hyper/J is a pure specification-based pointcut language.

Definition 3.4 (Progress-based Constructs). Let $J = \{0, \ldots, n\}$. A progress-based pointcut construct allows to select a join point $jp_i$ by reasoning on the following values:

$$val(v), v \in (\text{env}_{j})_{j \in J}.$$ 

However this definition – similar to state-based constructs – seems to be overly general. Again it seems reasonable that selection criteria only refer to elements of their respective join point context.

Definition 3.5 (Context- & Progress-based). Let $J = \{0, \ldots, n\}$. A progress-based pointcut construct allows to select a join point $jp_i$ by reasoning on values of join point contexts:

$$val(v), v \in (\text{context}(jp_{j}))_{j \in J}.$$ 

Another difference between pointcut constructs is, whether they permit to reason on past environments, or even future environments for a certain join point. The major difference is that the values provided by a previous environment are available, while values of future environments (in general) are not. Therefore, current aspect-oriented systems in general only provide a specific kind of progress-based pointcut constructs - past-based constructs.

Definition 3.6 (Past-based Constructs). A past-based pointcut construct allows to select a join point $jp_i$ by reasoning on values of join point contexts,

$$val(v), v \in (\text{context}(jp_{j}))_{j \in \{0, \ldots, i\}}.$$ 

While past-based pointcut constructs seem quite natural as all currently available progress-based constructs – like for example the $\text{cflow}$-pointcut in AspectJ or the proposed $\text{dataflow}$-pointcut – belong to this category; future-based constructs seem weird at first.

However one could also theoretically imagine a – very expensive – system allowing to set a save point when a future criterion has to be evaluated, resume normal execution until the specified criterion can be evaluated and if it evaluates to true reset to the save point and adapt the join point (and finally proceed with normal execution). As the practical value of these constructs can be doubted we restrict our discussion to past-based pointcut designators.

4. USING POINTCUT CONSTRUCTS TO CLASSIFY AOP SYSTEMS

The proposed classification of pointcut constructs can be directly used in order to classify aspect-oriented systems. In this section, we apply the classification to pointcut constructs to the AOP systems AspectJ, Hyper/J, Sally and AspectS.

4.1 Hyper/J

Hyper/J provides a pointcut language within its concern mapping, as well as in its hypermodules. Within the concern mapping file packages, classes, operations and fields are
produce a woven system. A weaver is modeled as a 11-tuple

 mechanisms as a weaver that combines two input programs to

is proposed. The framework models aspect-oriented mech-

 to compare aspect-oriented systems.

languages. However, there are works that provide frameworks

 to classify common characteristics of aspect-oriented systems, 

but the approach does provide distinguishing features that 

 can be used to compare different systems. For example, the 

modeling framework can be used to say that Hyper/J as 

well as AspectJ are aspect-oriented systems, but it is not 

possible to determine any distinguishing characteristics for 

them.

In different dynamic aspect-oriented systems are com-

pared. The underlying criteria for this comparison are for 

example efficiency or robustness. In contrast to the pro-

posed classification of pointcut language constructs, the cri-

teria used in are on a different level of abstraction. While 

the here proposed classification categorizes language con-

 structs according to the objects they reason on, the criteria 

used in concentrate on more implementation-specific is-

sues.

6. DISCUSSION AND CONCLUSION

In this paper we propose three classes of pointcut con-

 structs – specification-based, state-based and progress-based 

constructs. We applied these construct classes to a num-

ber of aspect-oriented systems (namely AspectJ, Hyper/J, 

Sally and AspectS) to illustrate that it is possible to ana-

lyze aspect-oriented systems in respect to their underlying 

pointcut languages. Each of the three proposed classes has 

a very different flavor associated with it.

6.1 Discussing the Construct Classes

First, we observe that state-based as well as progress-

based pointcut languages refer to a program’s state. Hence, 

both classes are somehow closer related to each other than 

to specification-based pointcut languages. The reason for 

this lies in the different abstraction underlying specification-

based, state-based and progress-based pointcut languages. 

While developers using a specification-based pointcut lan-

guage need to think in terms of syntactic elements of the 

base language, developers using a state- or progress-based 

pointcut language think in terms of objects and object-

relationships.

In our experience this syntax-based abstraction tends to 

be more complex to understand – in terms of the underlying 

pointcut semantics (although the set of selected join points 

might be easier to evaluate). Furthermore, developers typ-

ically have additional efforts to specify dynamic join point 

selection criteria in the adapting code as dynamic constructs 

are not available in the selection language.

For example, if the developer needs to intercept all mes-

sages (i.e. calls to a method) m from an object of type A in 

a class-based language like Java, he needs to specify all poten-

tial occurrences of m in class A as well as in all superclasses of 

A (for reasons of simplicity we neglect interfaces here). Since 

each occurrence of method calls in superclasses of A poten-
tially belong to an object which is not an instance of A, the developer needs to check within the join point adaptation whether or not the calling object actually is an instance of A for the currently selected static join point.

Second, we observe that it is under certain circumstances possible that an aspect-oriented system providing state-based but no past-based constructs permits to select the same join points. For example, if AspectJ would not provide the cflow construct, it is still possible to select all join points that occur in a certain control flow by providing additional implementation within the join point adaptation (manually maintain a call stack representation) and applying the if-construct in an appropriate way. Similar to the previous paragraph, this requires additional effort within the join point adaptation.

Although the difference between state-based and progress-based constructs does not seem to be that fundamental progress-based constructs indeed add language constructs which are not (or only with unreasonable efforts) expressible using state-based constructs. We outlined the dataflow pointcut designator as a prototypical example. However, this increased expressive power is not without cost: dataflow as well as cflow tend to be rather expensive constructs, their value is thus limited.

6.2 Conclusion

To summarize, the proposed construct classes provide an abstract understanding of aspect-oriented systems. From our point of view such an understanding is necessary in order to speak about different kinds of join point selections on a conceptual level without referring to concrete implementations.

We regard the construct classes as a useful tool to analyze future trends in aspect-oriented language development. Major research efforts are taken to provide new pointcut language constructs. Without an abstract understanding of the kind of language construct it is hardly possible to state how it relates to existing constructs.

Thus the proposed classes permit to better compare constructs on an abstract level and also suggest a design space for new constructs.

7. REFERENCES


