An Object-Based Model Representation System
Lending OO Features to Non-OO Modeling Languages

Gerd Frick, Eric Sax, Klaus D. Müller-Glaser
Forschungszentrum Informatik, Karlsruhe
Dept. of Electronic Systems and Microsystems
FZI/ESM
Haid-und-Neu-Str. 10-14, 76131 Karlsruhe, Germany
esm@fzi.de

Abstract

Reappearing structural decomposition principles for modeling embedded real-time systems are discussed and reduced to a simple object-based schema. An object structure calculus is developed which provides powerful object-oriented abstraction mechanisms. It founds the basis for a tool-independent model representation system which enhances standard CASE environments with features for differential development (reuse, variants). Further applications include model exchange, model integration and tool-independent code generation.

1. Introduction

Model-based development of embedded real-time software for industrial applications has grown adult following a structured analysis approach. Practically it is still in this stage, while state-of-the-art software development in general is heavily influenced by object-oriented principles.

Not all aspects attributed to object-orientation are equally applicable to embedded systems. Among the most important OO features lacking in currently used non-OO modeling languages is the notion of inheritance as a means of differential development. Developing a system by describing the differences to an existing system is an efficient way of describing variants and is obviously useful for other kinds of reuse. A ‘copy/paste/modify’ pattern of development is avoided. Shared parts are described only once and have to be maintained at a single place only – eliminating a potential source of inconsistencies.

In this paper we describe a model representation system which is tool (and language) independent and supports differential development in a powerful way (chapters 3 and 4). It is to be used as a backend technology in combination with existing and widely used CASE tools for embedded systems development (such as Statemate [10], Simulink [11], or ASCET-SD [4]) that do not support such features directly in their modelling languages. Chapter 2 is to motivate the design of our system.

2. System decomposition principles

Hierarchical decomposition is a well-established means of managing complexity (the ‘divide and conquer’ principle). A system is described as composed of and in terms of other (‘smaller’) systems. Embedded software systems resemble hardware systems in that the configuration of their (hierarchical) structure is static in most cases, i.e. at runtime there is no dynamic allocation of system components and bindings are static. Developers therefore are able to model an embedded control system in a constructive way in terms of individual objects and links between them. This contrasts with classes and associations declaratively describing relations between sets of objects, not which specific instance is bound to which other instances.

2.1. Entity structures

A common way of representing a network of collaborating objects is a graph-like structure of modules and channels with channels connected to modules by ports (modules are also called activities, actors, blocks, cells, capsules, and so on; channels are called flows, bindings, lines, signals, connectors, nets, etc.; ports are also called interfaces). Modules are the operating units while channels allow communication between the units. In hierarchically composed modules the ports at the boundary of the composite object are representatives of the outside world for the internal communication network.

Apart from activity and module charts in Statemate and block diagrams in Simulink and ASCET-SD this basic organizing principle is also found e.g. in SDL [15],
ROOM [16] (and its UML version [17]), the new SystemC 2.0 core language [18], the Koala component model [13] and generally in hardware and system-level design languages.

In SystemC 2.0 there are separate notions of ports and interfaces, where a channel implements one or more interfaces and a port accesses exactly one. A port connected to a channel must match one of the channel’s interfaces. The protocol roles in ROOM are roughly equivalent. At least the special case of two complementary interfaces/roles is supported by all other approaches: directed channels and input/output ports.

Channels may be as simple as a shared variable (with read and write interfaces) or an event object (with interfaces for signalling and observing) or may implement complex communication protocols.

Hierarchical decomposition clearly allows the refinement of modules in terms of inner modules and channels. On the other hand, complex communication behaviour can be also decomposed in the same way, i.e. a composite channel may contain internal channels as well as internal modules (e.g. storage cells for queuing messages).

Both modules and channels are essentially the same concept and can be unified to components having one or more interfaces. A port then is an internal proxy object bound to an interface of an external component. In SystemC 2.0 the components are still called modules, with a distinction between behaviour modules and channel modules. In the Koala component model, interfaces are called provides interfaces, ports are called requires interfaces.

The decomposition of a system into a hierarchy of (module or channel type) components, and bindings by means of interfaces and ports, shall in the following be called its entity structure.

### 2.2. Behavioral structures

The entity structure decomposes a system into functional units. A non-leaf module within the module hierarchy is implemented in terms of sub-modules. Many modelling languages enrich this ‘implemented in terms of’ by a notion of ‘dynamic’ behaviour in addition to the ‘static’ structural properties given by bindings between entities\(^1\). In effect, a complex unit of behaviour (a process) is associated with the module.

Atoms of behaviour may be the activation or deactivation of a module (i.e. of a process), sending a signal etc. Complex structures of behaviour are normally described by variants of finite state machines including those denoted in algorithmic form using e.g. if-then-else branches or while loops. State transitions are triggered implicitly by completion of a ‘previous’ process or explicitly by events and/or conditions on data variables. Elaborate graphical finite state machine formalisms like Harel’s statecharts [7] have hierarchical structures for states (and/or states) and transitions (compound transitions).

From a structural point of view, there is nothing special with behavioural structures. Like entity structures they work with hierarchical composition and bindings. The components are called statements, processes, function calls and the like; ports are more commonly called (formal) parameters here, while bindings to interfaces are more commonly known as actual parameters.

There are even bindings between entity components (e.g. variables, signals) and parameters of behavioural components (e.g. conditions, assignments, signal emissions). A separation of entity structures and behavioural structures in system descriptions is merely organizational like the one between controller and datapath in processor architectures [8].

The interpretation of entity structures vs. behavioural structures at runtime of course is different. This is much like the distinction in program execution between data and code, which are also physically the same in von-Neumann architectures.

Please note that structures must be finite in order to be described constructively, i.e. we cannot allow recursion (which describes structures declaratively by equations or equivalently as fixpoints of a function)\(^2\). Recursion in behavioural structures leads to dynamic allocation and binding of behavioural components (i.e. activation blocks on a stack). Since a recursive programming style is quite uncommon in the embedded systems domain (and is not included in the very common Statechart formalism) this restriction is acceptable. Please note that recursion is not only possible in behaviour (code) but also in entity (data) structures, which we also assume not to occur in embedded systems.

### 2.3. A tree of objects

After unification (from a structural point of view) of the notions of entity and behavioural structures a system is a hierarchically organized network of objects, where the network is defined by complementary interfaces and ports on objects (or ‘provides’ and ‘requires’ interfaces in Koala terminology; parameters and arguments or formal/actual parameters in programming terminology) and bindings.

---

\(^1\) Block diagrams for differential equations in continuous system modeling e.g. omit explicit behavioural structures because the computation model is fixed.

\(^2\) More precisely, we only have to disallow recursion which is not primitive recursion (as defined in the theory of computability). Primitive recursion knows the depth of recursion a priori.
For further simplification, interfaces can be considered as special cases of ordinary components of an object (namely components intended for ‘public’ access by a client). Two concepts remain: object and port. Ports are proxies for external objects and will more appropriately be called object references.

Objects are primitive objects or are composed of other objects and object references. A system therefore is an object tree where the leaf nodes are primitive objects or references. References pointing nowhere are unbound references. Another – equivalent – system view is a rooted directed graph where objects define nodes and containment and references define edges.

Entity decomposition can be stopped at the level of common primitive datatypes (like the set of integers or floating-point numbers) instead of refining them to an enumeration of their elements as primitive, referentiable objects. Therefore the resulting system is not purely object-based but also value-based, but values are restricted to appear only as ‘contents’ of leaf objects.

In summary, the essential building blocks of systems (or system models) have been reduced to a common basic structural principle – objects and references (or equivalently, nodes and directed edges). In chapter 3 we present an object structure calculus with features such as abstraction and inheritance that, applied to this problem domain, will be the basis of a model representation system that is able to combine the power of object-orientation (especially differential development) with traditional modeling languages and tools.

Entity decomposition can be stopped at the level of common primitive datatypes (like the set of integers or floating-point numbers) instead of refining them to an enumeration of their elements as primitive, referentiable objects. Therefore the resulting system is not purely object-based but also value-based, but values are restricted to appear only as ‘contents’ of leaf objects.

In summary, the essential building blocks of systems (or system models) have been reduced to a common basic structural principle – objects and references (or equivalently, nodes and directed edges). In chapter 3 we present an object structure calculus with features such as abstraction and inheritance that, applied to this problem domain, will be the basis of a model representation system that is able to combine the power of object-orientation (especially differential development) with traditional modeling languages and tools.

3. A pattern-based object structure calculus

This chapter presents an abstract language for the description of object structures as they have been motivated in the previous chapter. A thorough formal discussion is beyond the scope of this paper, though. A concrete implementation of this calculus can be based on XML [3], for example.

3.1. Objects

One of the most essential OO principles is object identity. Objects have a (possibly structured) content. Object identity makes an object different from another object (possibly structured) value such as a tuple. E.g. a pair (1.5, 2.75) represents a mathematically unique value while there may be many objects representing this value by their content.

More formally, an object \( \omega \) is a pair \((id_\omega, c_\omega)\) with an identity \(id_\omega\) and a content \(c_\omega\). The content is a collection \(\{\alpha_i\}\) of an arbitrary non-negative number of attributes. Object identities must be unique, therefore we need not distinguish between objects and their identities and can denote objects by \(id = \{\alpha_i\}\) where the equality sign denotes the binding between an object (identity) and the content.

An attribute \(\alpha\) is a pair \((l_\alpha, c_\alpha)\) with a label \(l_\alpha\) and a content \(c_\alpha\). The content is either void, a primitive value (data value) or an object. The labels must be unique among the attributes of an object, therefore we may identify attribute and label and again write \(l = c\). or “\(l\) is bound to \(c\)”, where the attribute label \(l\) must be qualified by an object, which is its scope. Attributes with void content are called unbound.

A primitive value is a character string, which can be used to encode any desired data value (character strings, integers, floating point numbers, Booleans etc.). The attributes of an object need not reference different objects, and cycles are allowed.

Viewed as a graph, objects and primitive values are nodes and attributes are directed edges. (Please note the equivalence to the model described in section 2.3.) If all objects are connected – which can be assured by introducing a root object with attributes bound to the roots of the otherwise unconnected partial graphs – we need no knowledge of object ids but can simply address all objects by the root object and paths through the attributes.

3.2. Object patterns

Objects are often associated with a notion of classes, where classes of objects are roughly the equivalent of types of values. More specifically, a class can be considered as a type for its members (or their content) and as a generator of new objects. But as there are untyped value systems (e.g. classical \(\lambda\)-calculus) there are also classless object systems. In such systems, called object-based in contrast to class-based, type systems and prototype objects (together with cloning) can be used to simulate classes. Object-based calculi are simpler and more flexible than traditional class-based calculi. (For a more in-depth discussion of class-based and object-based approaches see e.g. [1].) In our calculus we choose an object-based approach with a simple notion of object patterns for typing.

Object patterns are generalizations of objects. A pattern \(p\) is either
- the empty (or universal) pattern \(\emptyset\),
- the invalid (or bottom) pattern \(\perp\),
- a primitive pattern (a regular expression, which denotes a set of character strings),
- a generic object, which differs from an ordinary object in that attribute contents are patterns,
- an OR-term \(p \lor q\), which represents any value or object which is matched by one of the patterns,
- or an AND-term \( p \land q \), which represents any value or object which is matched by both of the patterns.

Please note that the empty pattern corresponds to void content, every primitive value is also a primitive pattern, and every object is also a generic object.

### 3.3. Pattern matching

We use a generalized notion of pattern matching where matching is not only between an instance and a pattern but also between patterns. Since every instance also is a pattern, we only need to define matching between patterns.

A pattern \( p \) is matched by (or is a sub-pattern of) a pattern \( q \), or \( q \) includes (or is larger than, or is a super-pattern of) \( p \), denoted \( p < q \), if any of the following conditions holds:

- \( p = q \),
- \( q \) is the empty (universal) pattern: \( p < \top \),
- \( p \) is the invalid pattern: \( \bot < q \),
- both \( p \) and \( q \) are primitive patterns (character strings, regular expressions) and regular expression pattern matching holds,
- both \( p \) and \( q \) are generic objects \( \{\ell_1 = p_1, \ldots, \ell_n = p_n\} \land \{m_1 = q_1, \ldots, m_k = q_k\} \)
  and \( \ell_j = m_j \Rightarrow p_i < q_j \).
- \( q \) is an OR-term \( q = r \lor s \) and either \( p < r \) or \( p < s \), or both,
- \( q \) is an AND-term \( q = r \land s \) and both \( p < r \) and \( p < s \).

Please note that \( < \) defines a partial order on patterns and has a single largest element, the empty pattern, and a single least element, the invalid pattern.

Pattern matching is sort of type checking for ‘instances’ (object against pattern) or ‘subtypes’ (pattern against pattern).

### 3.4. Reduction

OR-terms are sometimes and AND-terms are always reducible. A reducible term can be ‘evaluated’ and replaced by the result without affecting the set of objects matched by the pattern. The reduction rules actually define the semantics of the \( \lor \) and \( \land \) operators.

If a pattern \( p \) can thus be replaced by a pattern \( q \) we say \( p \) reduces to \( q \), or \( p \triangleright q \). Reduction is defined by the following set of rules (where \( p \) and \( p_i \) denote any kind of pattern, \( r \) and \( r_i \) denote regular expressions, and \( o \) and \( o_i \) denote generic objects):

\[
\begin{align*}
  p \lor p & \triangleright p, \\
  p \land p & \triangleright p, \\
  \top \lor p & \triangleright p, \\
  p \lor \bot & \triangleright p, \\
  p \land \bot & \triangleright \bot, \\
  \bot \lor \bot & \triangleright \bot, \\
  r_i r_i & \triangleright r_1 \land r_2, \\
  r_i r_i & \triangleright r_2 \land r_1, \\
  \bot & \land \bot, \\
  r_i r_i & \triangleright r_i \land \bot, \\
  \bot & \land \bot, \\
  r_i r_i \lor o_i & \triangleright \text{UNION}(o_i, o_i), \\
  p \lor (p_2 \lor p_3) & \triangleright (p_1 \lor p_2) \lor (p_1 \lor p_3), \\
  (p_1 \lor p_2) \lor p_3 & \triangleright (p_1 \lor p_3) \lor (p_2 \lor p_3),
\end{align*}
\]

where \( \text{UNION} \) is defined as follows:

\[
\text{UNION}(\{l_i = p_i\} \cup \{m_i = q_i\}) = \{l'_i = p'_i\} \cup \{m'_i = q'_i\},
\]

with

\[
\begin{align*}
  l'_i = l_j & \Rightarrow m'_i = m_l, \\
  l'_i = l_j & \Rightarrow p'_i = p_j \land q_k,
\end{align*}
\]

and the patterns on unshared labels remaining the same.

In other words, the result of the \( \text{UNION} \) operation on object contents is the union of all attributes where in case of label collision the patterns are combined with an AND-operator.

The \( \triangleright \) relation can easily be extended to patterns in general where a sub-term is reduced according to one of the rules and to the closure over sequences of one-step reductions. We omit the proof that the reduction rules are well-defined, i.e. do not contradict the definition of \( < \), and that \( \land \) and \( \lor \) are associative and commutative.

### 3.5. Interpretation in OO terminology

The basic elements of the calculus are values, objects and references (bindings of attributes of objects). A notion of types is introduced by patterns which are expressions over values and objects.

Primitive patterns (regular expressions for character strings) can be interpreted as primitive data types.
Generic objects take the role of object types. They define a required set of attributes and type constraints for the attributes.

OR-terms introduce polymorphism. A value or object can match any of the patterns combined by the \( \lor \) operator. The closure of OR composition is the universal pattern (any type).

AND-terms provide specialization by inheritance, extension and restriction. They deserve some more explanation:

Beyond the basic assumption of object identities (and leaving aside the encapsulation of functions and data, which are unified in our system) one of the most important features of OO techniques is the built-in support of differential development by the notion of inheritance. Inheritance is often paired with a notion of subclassing or specialization (‘is-a’ relationship between classes; corresponding to a subset relationship when classes are viewed as sets of objects) which can be transferred to object patterns. The sub-pattern relationship takes this role.

Inheritance as known from popular class-based languages (such as C++, Java, or UML) is a means of specialization by extension. It is often complemented by some possibilities of overriding, i.e. replacing inherited properties (virtual methods are best-known). Overriding as implemented in languages like C++ has to be restricted carefully in order to maintain is-a semantics [12].

Overriding in general should provide means for specialization by restriction. Unfortunately this variant of specialization is not equally well-supported in current OO modeling (UML) and programming languages. Nevertheless, the case for it appears frequently in specialization hierarchies.

Consider e.g. a class A with subclasses B and C which is associated with a class X with subclasses Y and Z. Assume that concerning the association between As and Xs we know that Bs are only linked with Ys and Cs with Zs. There is no easy way in UML to model this specialization of the association between A and X along their subclass hierarchies.

Both types of specialization are covered equally well by the \( \land \) operator. As defined on generic objects an AND-term after reduction contains the union set of the attributes of both operands where in case of label collision the patterns are combined with an AND-operator. Practically, one operand is an existing generic object (the superclass) while the other operand is the difference to a new generic object (the subclass). The difference realizes extension by including new labels and restriction by providing pattern restrictions on existing labels. Please note that in case of label collision both attribute patterns are combined by an \( \land \) operator which means that the type of the attribute is restricted to a subtype. (Future extensions of this calculus may include a NOT operator for additional restriction power.)

Resuming the previously mentioned example, let there be patterns \( X, Y, Z \) with \( X = Y \lor Z \) and \( Y \land Z \neq \perp \) and let a generic object \( A = \{ a = X, b = X \} \) be extended by a difference pattern \( B' = \{ b = Y, c = Z \} \) to a generic object \( B = A \land B' \). Reduction then yields \( B = \{ a = X, b = Y, c = Z \} \) showing that the attribute \( b \) is restricted from \( X \) to \( Y \) along the specialization from \( A \) to \( B \).

Concerning the notion of specialization, an object of the result type (subclass; \( B \) in the example) is-an object of both the original type (superclass; \( A \) ) and the difference type (mixin class; \( B' \)). In pattern terminology the resulting pattern is a sub-pattern of either input pattern (\( B \prec A \) and \( B \prec B' \)). Multiple inheritance can be realized by composition of multiple AND-terms. The closure of AND composition is the invalid pattern (overspecialized, no object matched).

Subtyping is not the only purpose of specialization by the \( \land \) operator. Instantiation works exactly the same way. Wherever the pattern leaves a degree of freedom (e.g. of a data value by a regular expression or of an object structure by an OR-term) a mixed-in difference can choose the specific instance. An object pattern without any remaining degrees of freedom is an ordinary object, whereas an object pattern with variable parts is an object template (class).

Viewed this way the pattern mechanism is an abstraction mechanism, while refinement by AND composition is the corresponding application mechanism. Subtyping is partial application, instantiation is full application. Simply assigning the content pattern of an existing object to a new object is cloning. By the described unification of refinement and instantiation the \( \prec \) relation gets a unified ‘is-a’ meaning.

Please note that patterns are described in the same language as objects are (actually objects use a subset of the language) and can be refined uniformly to ordinary objects within the same syntax: there is no additional schema language or the like. Furthermore note that in an object-based language like this calculus there is no need of singleton classes; they are simply described as ordinary objects.

3.6. Comparison with existing approaches

The major OO principles covered by our object structure calculus and discussed in the previous section (object identity, inheritance, polymorphism etc.) are well-known. Object calculi have been studied in depth by e.g. Abadi and Cardelli [1].

Patterns, too, have a long tradition on its own, at least for text processing and functional programming, and have
recently been applied also to XML processing [9]. Patterns as designed and used in our calculus resemble to some extent the type systems found in (functional) programming languages: AND and OR patterns are comparable to product and sum types [14].

Our approach tries to combine some useful pieces of good practice in language design to form a simple but powerful calculus as the basis for the design of a special purpose language (the special purpose is the model representation system described in the next chapter). Furthermore we address some problems solved not satisfactorily in currently popular OO languages like UML (especially specialization by restriction; also the diamond problem for multiple inheritance has been avoided in an elegant way by the AND logic of our inheritance operation). The main difference, though, from other object calculi or OO languages is the strict avoidance of a built-in notion of behaviour. Interpretation of the structures described by the calculus is delegated to some externally defined semantics (see chapter 4). We consider it a matter of ‘separation of concerns’. The achieved advantage is the simplicity and at the same time wide applicability of the calculus.

4. An object-based model representation system and its applications

The object structure calculus described in the previous chapter has been aligned with the common system decomposition principles extracted from modeling languages as described in chapter 2. It serves as the basis for a generic object-oriented representation system for models of embedded real-time systems including those modeled with traditional non-OO modeling languages and tools.

The main idea is to factorize existing modeling languages towards two orthogonal parts:

1. The basic semantic primitives, such as states and transitions (control flow), data stores and data flows, built-in operators etc.

2. Hierarchy and abstraction operators, such as subsystems, superblocks, masks, generic charts, functions, classes etc.

For (1), a base library is built up, organized by the principles developed in chapter 2. It consists of semantic primitives in a specialization hierarchy that follows the paths of unification in reverse order. Some library elements are tagged as being equivalent to some semantic primitives of existing modeling languages (external semantics). The operators elicited in (2) are replaced by equivalents in the object structure calculus of chapter 3.

Models originating from various modeling languages can be represented uniformly in this system. As far as the library created in (1) unifies comparable primitives of different languages the representation system can serve as a channel for model exchange between modeling languages (tools). Perhaps more important is the potential gained by (2). The power of genericity and differential development obtained by the object pattern calculus excels the ‘native’ abstraction mechanisms built into the modeling languages of state-of-the-art CASE tools:

- Simulink from The Mathworks [11] uses subsystems for hierarchy, masked subsystems for abstraction; it has an object-based (not class-based) block library concept but has no notion of specialization/inheritance.

- Statemate from I-Logix [10] employs hierarchical chart types, uses generic charts for abstraction and also lacks of specialization/inheritance.

- ASCET-SD from ETAS [4] offers module hierarchies without abstraction or specialization/inheritance, followed by an object hierarchy with abstraction by classes but no specialization/inheritance as well.

Models transferred from a (front-end) modeling language, implemented by a CASE tool, to the (back-end) model representation system, implemented e.g. by a CASE repository, can be used to derive other models differentially within the general representation (appropriate structure editors presupposed). The reduction system of the calculus (section 3.4) always allows us to flatten inheritance hierarchies, which is necessary for transferring the models back to a CASE tool. Thus the developer is provided with additional tools for model reuse which is specially important for the development of variants or product families.

Further applications include:

- Integration of subsystems, even if developed with different tools,

- The option of implementing a tool-independent (rapid prototyping or production) code generator which directly reads from the CASE repository.

5. Conclusion and future work

The essential system decomposition principles found in existing modeling languages for embedded real-time systems can be reduced to a common basic structural schema – objects and references (or equivalently, nodes and directed edges) – which applies to both entity and behavioural structures. By a library-based translation to a simple, object-based structure calculus, less powerful...
abstraction mechanisms can be replaced by a pattern-based one which provides polymorphism and specialization (subtyping). Reduction within the calculus allows back-translation to a modeling language.

An object-based model representation system (e.g. a CASE repository) implementing this approach thus lends OO features to non-OO modeling languages. It supports differential development, which is of special importance for variants and product lines. Further applications include model exchange, model integration and code generation.

In our ongoing research we are reducing the object structure calculus described in this paper to classical λ-calculus in order to clarify the relationship to one of the foundational theories of computer science and to leverage the power of functional abstraction for a modular implementation of the model representation system.

6. References


4 Currently, a pre-version of such a system is under (prototype) development as part of an industrial project together with Bosch and SME-companies. This project is sponsored by the German state ‘Baden-Württemberg’ (‘Zukunftsoffensive Junge Generation’, project ‘COMTESSA’).