*5*

## Fundamental Paradigms

*Every DSL is different. It is driven by the domain for which it is built. However, as it turns out, there are also a number of commonalities between DSLs. These can be handled by modularizing and reusing (parts of) DSLs, as discussed in the last section of the previous chapter. In this section we look at common paradigms for describing DSL structure and behavior.*

### 5.1 Structure

Languages have to provide a means of structuring large programs in order to keep them manageable. Such means include modularization and encapsulation, specification vs. implementation, specialization, types and instances, as well as partitioning.

|  |  |
| --- | --- |
| *5.1.1 Modularization and Visibility*  DSLs often provide some kind of logical unit structure, such as namespaces or modules. Visibility of symbols may be restricted to the same unit, or to referencing ("importing") units. Symbols may be declared as **public** or **private**, the latter making them invisible to other modules, which guarantees that |  |
| changes to these symbols cannot affect other modules. Some |  |

form of namespaces and visibility is necessary in almost any DSL. Often there are domain concepts that can play the role of the module, possibly oriented towards the structure of the organization in which the DSL is used.

**mbeddr C:** As a fundamental extension to C, this DSL contains modules with visibility specifications and imports. Functions, state machines, tasks and all other top-level

concepts reside in modules. Header files (which are effectively a poor way of managing symbol visibility) are only used in the generated low-level code and are not relevant to the user of mbeddr C. J

**Component Architecture:** Components and interfaces live in namespaces. Components are implementation units, and are always private. Interfaces and data types may be public or private. Namespaces can import each other, making the public elements of the imported namespace visible to the importing namespace. The OSGi generator creates two different bundles: an interface bundle that contains the public artifacts, and an implementation bundle with the components. In the case of a distributed system, only the interface bundle is deployed on the client. J

**Pension Plans:** Pension plans constitute namespaces. They are grouped into more coarse-grained packages that are aligned with the structure of the pension insurance business. J

#### 5.1.2 Partitioning

Partitioning refers to the breaking down of programs into several physical units such as files (typically each model fragment is stored in its own partition). These physical units do not have to correspond to the logical modularization of the models within the partitions. For example, in Java a public class has to live in a file of the same name (logical module == physical partition), whereas in C# there is no relationship between namespace, class names and the physical file and directory structure. A similar relationship exists between partitions and viewpoints, although in most cases, different viewpoints are stored in different partitions.

Note that a reference to an element should not take into account the partition in which the target element lives. Instead, it should only use the logical structure. Consider an element **E** that lives in a namespace **x.y**, stored in a partition **mainmodel**. A reference to that element should be expressed as **x.y.E**, not as **mainmodel.E** or **mainmodel/x.y.E**. This is important, as it allows elements to move freely between partitions without this leading to updates of all references to the element.

Partitioning may have consequences for language design.

Consider a textual DSL in which a concept A contains a list of

instances of concept B. The B instances then have to be physically nested within an instance of A in the concrete syntax. If there are many instances of B in a given model, they cannot be split into several files, so these files may become big and result in performance problems. If such a split must be possible, this has to be designed into the language.

**Component Architecture:** A variant of this DSL that was used in another project had to be changed to allow a namespace to be spread over several files for reasons of scalability and version-control granularity. In the initial version, namespaces actually *contained* the components and interfaces. In the revised version, components and interfaces were owned by no other element, but model files (partitions) had a namespace declaration at the top, logically putting all the contained interfaces and components into this namespace. Since there was no technical containment relationship between namespaces and their elements, several files could now declare the same namespace. Changing this design decision lead to a significant reimplementation effort, because all kinds of naming and scoping strategies changed. J

Other concerns influence the design of a partitioning strategy as well:

*Change Impact* Which partition changes as a consequence of a particular change of the model (changing an element name might require changes to all references to that element from other partitions).

*Link Storage* Where are links stored (are they always stored in the model that logically "points to" another one?), and if not, how/where/when to control reference/link storage.

*Model Organization* Partitions may be used as a way of organizing the overall model. This is particularly important if the tool does not provide a good means of presenting the overall logical structure of models and finding elements by name and type. Organizing files with meaningful names in directory structures is a workable alternative.

*Tool Chain Integration* Integration with existing, file-based tool chains. Files may be the unit of check in/check out, versioning, branching or permission checking.

It is often useful to ensure that each partition is processable separately to reduce processing times. An alternative approach supports the explicit definition of those partitions that should

be processed in a given processor run (or at least a search path, a set of directories, to find the partitions, like an include path in C compilers or the Java classpath). You might even consider a separate build step to combine the results created from the separate processing steps of the various partitions (again like a C compiler, which compiles every file separately into an object file, after which the linker handles overall symbol/reference resolution and binding).

The partitioning scheme may also influence users’ team collaboration when editing models. There are two major collaboration models: real-time and commit-based. In real-time collaboration, a user sees his model change in real time as another user changes the same model. Change propagation is immediate. A database-backed repository is often a good choice regarding storage, since the granularity tracked by the repository is the model element. In this case, the partitioning may not be visible to the end user, since they just work "on the repository". This approach is often (at least initially) preferred by non-programmer DSL users.

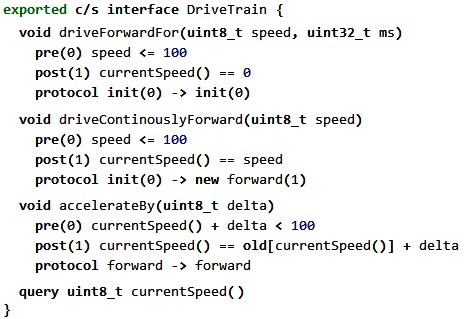
The other collaboration mode is commit-based, in which a user’s changes are only propagated to the repository if he performs a *commit*, and incoming changes are only visible after a user has performed an *update*. While this approach can be used with database-backed repositories, it is most often used with file-based storage. In this case, the partitioning scheme is visible to DSL users, because it is those files they commit or update. This approach tends to be preferred by developers, maybe because well-known versioning tools have used the approach for a long time.

#### 5.1.3 Specification vs. Implementation

Separating specification and implementation supports plugging in different implementations for the same specification and hence provides a way to "decouple the outside from the inside"[[1]](#footnote-1). This supports the exchange of several implementations behind a single interface. This is often required as a consequence of the development process: one stakeholder defines the specification and a client, whereas another stakeholder provides one or more implementations.

A challenge for this approach is how to ensure that all implementations are consistent with the specification. Traditionally, only the structural/syntactic/signature compatibility is checked. To ensure semantic compatibility, additional means that specify the expected *behavior* are required. This can be achieved with pre- or post-conditions, invariants or protocol state machines.

**mbeddr C:** This DSL adds interfaces and components to C. Components provide or use one or more interfaces. Different components can be plugged in behind the same interface. To support semantic specifications, the interfaces support pre- and post-conditions as well as protocol state machines. Fig. 5.1 shows an example. Although these specifications are attached to interfaces, they are actually checked (at runtime) for all components that provide the respective interface. J



**Refrigerators:** Cooling programs can access hardware elements (compressors, fans, valves); those are defined as part of the refrigerator hardware definition. To enable cooling programs to run with different, but similar hardware configurations, the hardware structure can use "trait inheritance", by which a hardware trait defines a set of hardware elements, acting as a kind of interface. Other hardware configurations can inherit these traits. As long as cooling programs are only written against traits, they work with any refrigerator that implements the particular set of traits against which the program is written. J

#### 5.1.4 Specialization

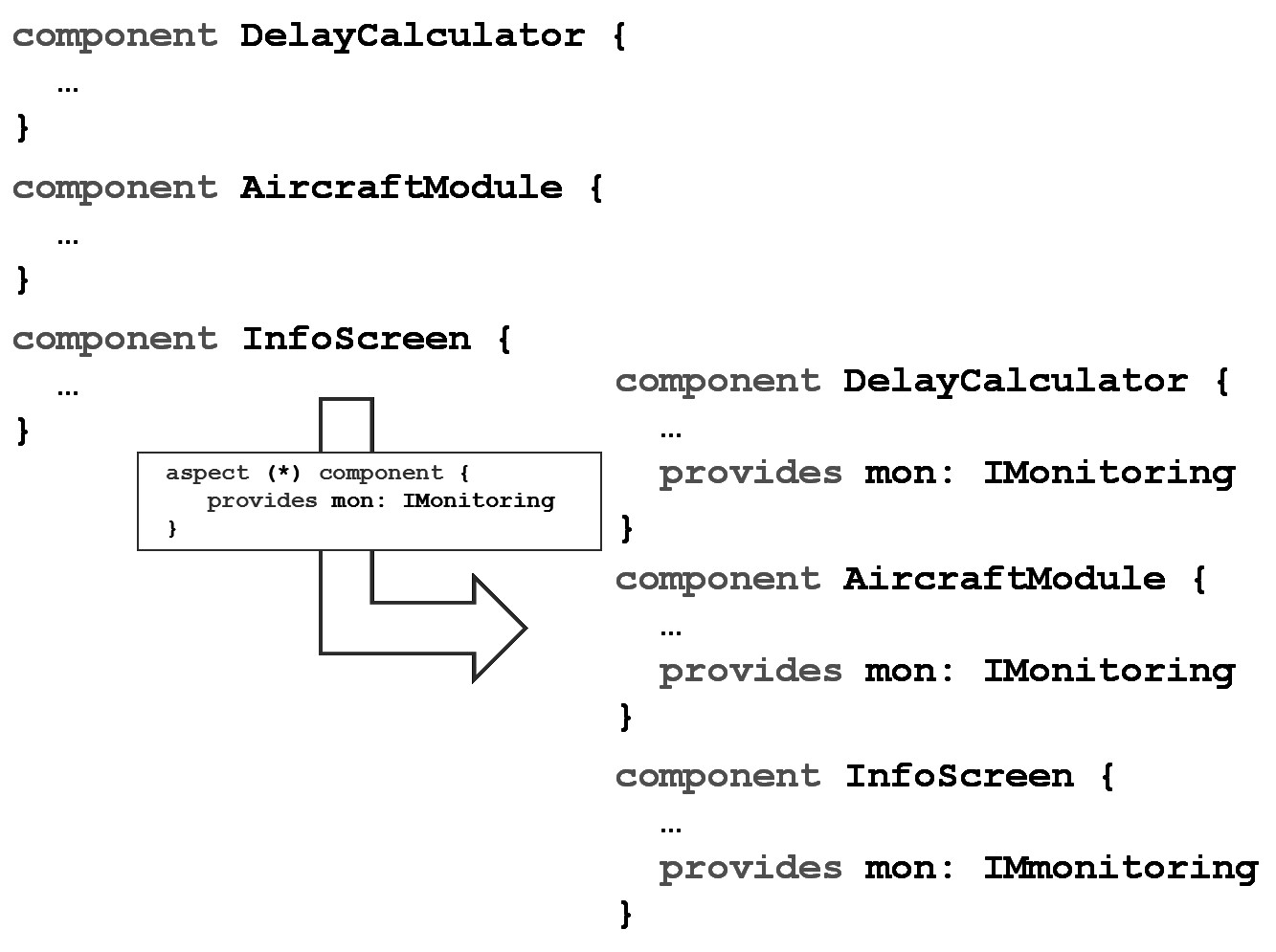
|  |  |
| --- | --- |
|  |  |
| cialization in the context of DSLs can be used for implementing variants or for evolving a program over time.  Defining the semantics of inheritance for domain-specific language concepts is not always easy. The various approaches found in programming languages, as well as the fact that some of them lead to problems (multiple inheritance, diamond inheritance, linearization, or code duplication in Java’s interface inheritance) shows that this is not a trivial topic. It is a good idea to just copy a suitable approach *completely* from a programming language in which inheritance seems to work well. Even small changes can make the whole approach inconsistent.  **Pension Plans:** The customer using this DSL had the challenge of creating a huge set of pension plans, implementing changes in relevant law over time, or implementing related plans for different customer groups. Copying complete plans and then making adaptations was not feasible, because this resulted in a maintenance nightmare: a large number of similar but not identical pension plans. Hence the DSL provides a way for pension plans to inherit from one another. Calculation rules can be marked *abstract* (needing to be overwritten in sub-plans), *final* rules are not overwritable. Visibility modifiers control which rules are considered "implementation details". J  **Refrigerators:** A similar approach is used in the cooling DSL. Cooling programs can specialize other cooling programs. Since the programs are fundamentally state-based, we had to define what it means to specialize a cooling program: a subprogram can add additional event handlers and transitions to states. New states can be added, but states defined in the super-program cannot be removed. J |  |
| *5.1.5 Types and Instances*  Types and instances supports the definition of structures that can be parametrized upon instantiation. This allows reuse of |  |
| common parts, and expressing variability via parameters. |  |

Specialization enables one entity to be a more specific variant of another. Typically, the more specific one can be used in all contexts in which the more general one is expected (the Liskov substitution principle2). The more general one may be incom-

**mbeddr C:** Apart from C’s **structs** (which are instantiatable data structures) and components (which can be instantiated and connected), state machines can be instantiated as well. Each instance can be in a different state at any given time. J

#### 5.1.6 Superposition and Aspects

|  |  |
| --- | --- |
| Superposition refers to the ability to merge several model frag- |  |
| ments according to some DSL-specific merge operator. Aspects provide a way of "pointing to" several locations in a program based on a pointcut operator (essentially a query over a program or its execution), adapting the model in ways specified by the aspect. Both approaches support the compositional creation of many different model variants from the same set of model fragments.  **Component Architecture:** This DSL provides a way of advising component definitions from an aspect (Fig. 5.2). An aspect may introduce an additional **provided port mon: IMonitoring** that allows a central monitoring component to query the advised components via the **IMonitoring** interface. J |  |



**WebDSL:** Entity declarations can be *extended* in separate modules. This makes it possible to declare in one module all data declarations of a particular feature. For example, in the *researchr* application, a **Publication** can be **Tag**ged, which requires an extension of the **Publication** entity. This extension is defined in the **tag** module, together with the definition of the **Tag** entity. This is essentially a use of superposition. J

#### 5.1.7 Versioning

Often, variability over time of elements in DSL programs has to be tracked. One alternative is to simply version the model files using existing version control systems, or the version control mechanism built into the language workbench. However, this requires users to interact with often complex version control systems and prevents domain-specific adaptations of the version control strategy.

The other alternative is to make versioning and tracking over time a part of the language. For example, model elements may be tagged with version numbers, or specify a revision chain by pointing to a previous revision, enforcing compatibility constraints between those revisions. Instead of declaring explicit versions, business data is often time-dependent, where different revisions of a business rule apply to different periods of time. Support for these approaches can be built directly into the DSL, with various levels of tool support.

**mbeddr C:** No versioning is defined into the DSL. Users work with MPS’ integration with popular version control systems. Since this DSL is intended for use by programmers, working with existing version control systems is not a problem. J

**Component Architecture:** Components can specify a **new version of** reference to another component. In this case, the new version may specify additional provided ports with the same interfaces, or with new versions of these interfaces. The new version may also deprecate required ports. Effectively, this means that the new version of something must be replacement-compatible with the old version (the Liskov substitution principle again). J

**Pension Plans:** In the pension workbench, calculation rules declare applicability periods. This supports the evolution of calculation rules over time, while retaining reproducability for calculations performed at an earlier point in time. Since the Intentional Domain Workbench is a projectional tool, pension plans can be shown with only the version of a rule that is valid for a given point in time. J

### 5.2 Behavior

|  |  |
| --- | --- |
| ments, or simply by interacting with domain-specific structures or data.  Note that there are two kinds of DSLs that don’t make use of these kinds of behavior descriptions. Some DSLs really just specify structures. Examples include data definition languages or component description languages (although both of them often use expressions for derived data, data validation or preand post-conditions). Other DSLs specify a set of expectations regarding some behavior (declaratively), and the generator creates the algorithmic implementation. For example, a DSL may specify, simply with a tag such as *async*, that the communication between two components shall be asynchronous. The generator then maps this to an implementation that behaves according to this specification.  **Component Architecture:** The component architecture DSL is an example of a structure-only DSL, since it only describes black box components and their interfaces and relationships. It uses the specification-only approach to specify whether a component port is intended for synchronous or asynchronous communication. J  **mbeddr C:** The component extension provides a similar notion of interfaces, ports and components as in the previous example. However, since here they are directly integrated with C, C expression can be used for pre- and post-conditions of interface operations (see Fig. 5.1). J  Using an established behavioral paradigm for a DSL has sev- |  |
| eral advantages4. First, it is not necessarily simple to define |  |

The behavior expressed with a DSL must of course be aligned with the needs of the domain. However, in many cases, the behavior required for a domain can be derived from well-known behavioral paradigms3, with slight adaptations or enhance-

consistent and correct semantics in the first place. By reusing an existing paradigm, one can learn about advantages and drawbacks from existing experience. Second, a paradigm may already come with existing means for performing interesting

analyses (as in model checking or SMT solving) that can easily be used to analyse DSL programs. Third, there may be existing generators from a behavioral paradigm to an efficient executable for a given platform (state machines are a prime candidate). By generating a model in a formalism for which such a generator exists, we reduce the effort for building an end-to-end generator. If our DSL uses the same behavioral paradigm as the language for which the generator exists, writing the necessary transformation is straightforward (from a semantic point of view).

The last point emphasizes that using an existing paradigm for a DSL (e.g. state-based) does not mean that the concepts have to directly use the abstractions used by that paradigm (just because a program is state-based does not mean that the concept that acts as a state has to be called state, etc.).

. This section describes some of the most well-known behavioral paradigms that can serve as useful starting points for behavior descriptions in DSLs. In addition to describing the paradigm, we also briefly investigate how easily programs using the paradigm can be analyzed, and how complicated it is

to build debuggers.

#### 5.2.1 Imperative

Imperative programs consist of a sequence of statements, or instructions, that change the state of the program. This state may be local to some kind of module (e.g., a procedure or an object), global (as in global variables) or external (when communicating with peripheral devices). Procedural and object-oriented programming are both imperative, using different means for structuring and (in the case of OO) specialization. Because of aliasing and side effects, imperative programs are expensive to analyse. Debugging imperative programs is straightforward and involves stepping through the instructions and watching the state change.

**mbeddr C:** Since C is used as a base language, this language is fundamentally imperative. Some of the DSLs on top of it use other paradigms (the state machine extension is state-based, for example). J

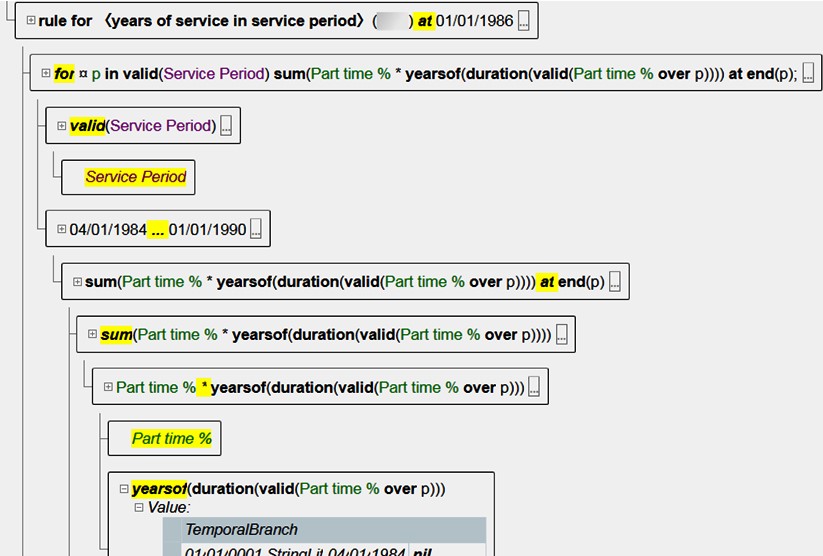
**Refrigerators:** The cooling language integrates various paradigms, but contains sequences of statements to implement aspects of the overall cooling behavior. J

|  |
| --- |
| *5.2.2 Functional*  Functional programming uses functions as the core abstraction. In purely functional programming, a function’s return value only depends on the values of its arguments. Calling the same function several times with the same argument values returns the same result (that value may even be cached!). Functions cannot access global mutable state, no side effects are allowed. These characteristics make functional programs very easy to analyze and optimize. These same characteristics, however, also make purely functional programming relatively useless, because it cannot affect its environment (after all, this would be a side effect). So, functional programming is often only used for parts ("calculation core") of an overall program and integrates with, for example, an imperative part that deals with IO.  Since there is no changing state to observe as the program steps through instructions, debugging can be done by simply showing all intermediate results of all function calls as some kind of tree, basically "inspecting" the state of the calculation. This makes building debuggers relatively simple.  **Pension Plans:** The calculation core of pension rules is functional. Consequently, a debugger has been implemented that, for a given set of input data, shows the rules as a tree that shows all intermediate results of each function call  (Fig. 5.3). No "step through" debugger is necessary. J  Pure expressions are an important subset of functional programming (as in **i > 3\*2 + 7**). Instead of calling functions, operators are used. However, operators are just infix notations for function calls. Usually the operators are hard wired into the language and it is not possible for users to define their own functional abstractions. The latter is the main differentiator to functional programming in general. It also limits expressivity, since it is not possible to modularize an expression or to reuse expressions by packaging into a user-defined function. Consequently, only relatively simply tasks can be addressed with a pure expression language[[2]](#footnote-2). |

**mbeddr C:** We use expressions in the guard conditions of the state machine extension as well as in pre- and postconditions for interface operations. In both cases it is not possible to define or call external functions. Of course, (a subset of) C’s expression language is reused here. J

since the solution algorithm may be very complex and possibly not even be known to the user of the language.

|  |  |
| --- | --- |
|  |  |
| *5.2.3 Declarative*  Declarative programming can be considered the opposite of imperative programming (and, to some extent, of functional programming). A declarative program does not specify any control flow; it does not specify a sequence of steps of a calculation. A declarative program only specifies *what* the program should accomplish, not *how*. This is often achieved by specifying a set of properties, equations, relationships or constraints. |  |
| Some kind of evaluation engine then tries to find solutions. The particular advantage of this approach is that it does not predefine how a solution is found; the evaluation engine has a lot of freedom in doing so, possibly using different approaches in different environments, or evolving the approach over time6. This large degree of freedom often makes finding the solution |  |
| expensive – trial and error, backtracking or exhaustive search may be used7. Debugging declarative programs can be hard, |  |

.

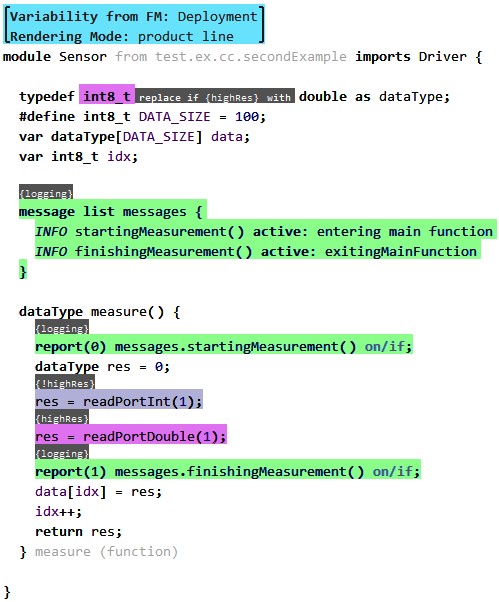
Declarative programming has many important subgroups. For *concurrent programs*, a declarative approach allows the efficient execution of a single program on different parallel hardware structures. The compiler or runtime system allocates the program to available computational resources. In *constraint programming*, the programmer specifies constraints between a set of variables. The engine tries to find values for these variables that satisfy all constraints. Solving mathematical equation systems is an example, as is solving sets of Boolean logic formulas. *Logic programming* is another sub-paradigm, in which users specify logic clauses (facts and relations) as well as queries. A theorem prover then tries to solve the queries.

**Component Architecture:** This DSL specifies timing and resource characteristics for component and interface operations. Based on this data, one could run an algorithm which allocates the component instances to computing hardware so that the hardware is used as efficiently as possible, while at the same time reducing the amount of network traffic. This is an example of constraint solving used to synthesize a schedule. J

**mbeddr C:** This DSL supports presence conditions for product line engineering. A presence condition is a Boolean expression over a set of configuration features that determines whether the associated piece of code is present for a given combination of feature selections (Fig. 5.4). To verify the structural integrity of programs in the face of varying feature combinations, constraint programming is used (to ensure that there is no configuration of the program in which a reference to a symbol is included, but the referenced symbol is not). A set of Boolean equations is generated from the program and the attached presence conditions, . A solver then makes sure they are consistent by trying to find an example solution that violates the Boolean equations. J

**Example:** The Yakindu DAMOS block diagram editor supports custom block implementation based on the Mscript language (Section 5.5). It supports declarative specification of equations between input and output parameters of a block. A solver computes a closed, sequential solution that efficiently calculates the output of an overall block diagram. J

.

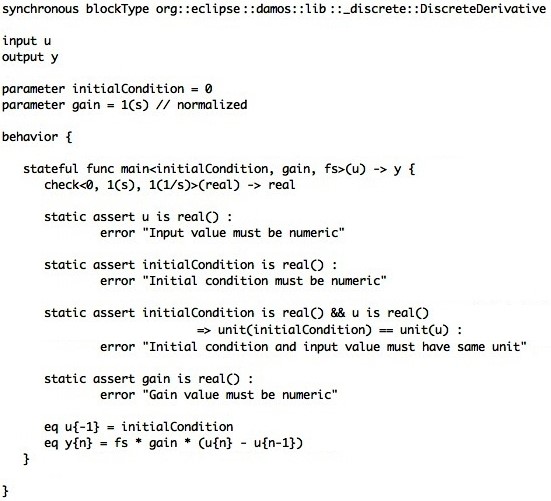


**Example:** Another example for declarative programming is the type system DSL used by MPS itself. Language developers specify a set of type equations containing free type variables, among other things. A unification engine tries to solve the set of equations by assigning actual types to the free type variables so that the set of equations is consistent. We describe this approach in detail in Section 10.4. J

#### 5.2.4 Reactive/Event-based/Agent

In this paradigm, behavior is triggered based on received events.

Events may be created by another entity or by the environment (through a device driver). Reactions are expressed by the creation of other events. Events may be globally visible or explicitly routed between entities, possibly using filters and/or using priority queues. This approach is often used in embedded systems that have to interact with the real world, where the real



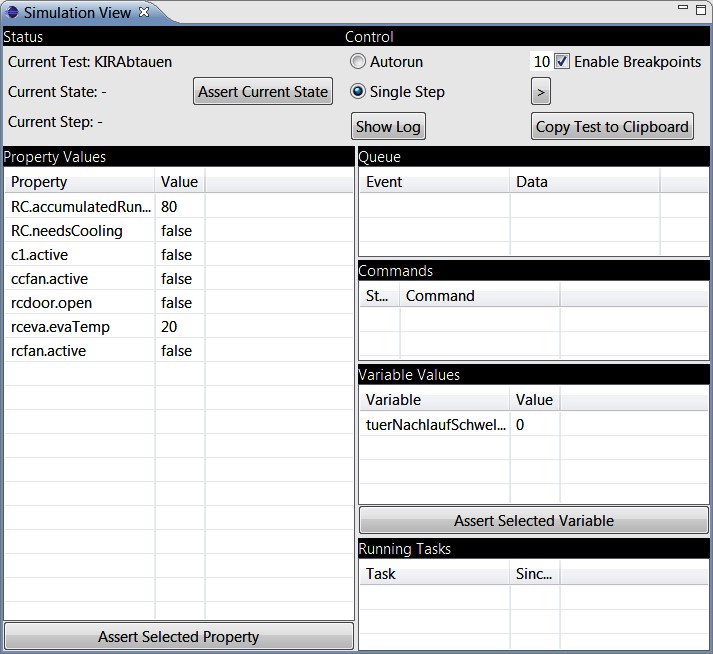
world produces events as it changes. A variant of this approach queries input signals at intervals controlled by a scheduler and considers changes in input signals as the events.

**Refrigerators:** The cooling algorithms are reactive programs that control the cooling hardware based on environment events. Such events include the opening of a refrigerator door, the crossing of a temperature threshold, or a timeout that triggers defrosting of a cooling compartment. Events are queued, and the queues are processed in intervals determined by a scheduler. J

Debugging is simple if the timing/frequency of input events can be controlled. Visualizing incoming events and the code that is triggered as a reaction is relatively simple. If the timing of input events cannot be controlled, then debugging can be almost impossible, because humans are much too slow to fit "in between" events that may be generated by the environment in rapid succession. For this reason, various kinds of simulators are used to debug the behavior of reactive systems, and sophisticated diagnostics regarding event frequencies or queue filling levels may have to be integrated into the programs as they run in the real environment.

**Refrigerators:** The cooling language comes with a simula-

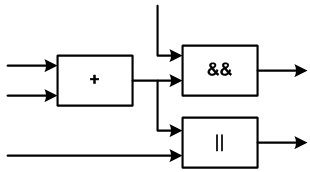
tor (Fig. 5.6) based on an interpreter in which the behavior of a cooling algorithm can be debugged. Events are explicitly created by the user, on a timescale that is compatible with the debugging process. J



#### 5.2.5 Dataflow

The dataflow paradigm is centered around variables with dependencies (in terms of calculation rules) among them. As a variable changes, the variables that depend on the changing variable are recalculated. We know this approach mainly from two use cases. One is spreadsheets: cell formulas express dependencies to other cells. As the values in these other cells change, the dependent cells are updated. The other use case is data flow (or block) diagrams (Fig. 5.7), used in embedded software, extraction-transfer-load data processing systems and enterprise messaging/complex event processing. There, the calculations or transformations are encapsulated in the blocks, and the lines represent dependencies – the output of one blocks "flows" into the input slot of another block. There are three different execution modes:

* The first one considers the data values as continuous sig-



nals. At the time one of the inputs changes, all dependent values are recalculated. The change triggers the recalculation, and the recalculation ripples through the dependency graph. This is the model used in spreadsheets.

* The second one considers the data values as quantized, unique messages. A new output message is calculated only if a message is available for all inputs. The recalculation synchronizes on the availability of a message at each input, and upon recalculation, these messages are consumed. This approach is often used in ETL and CEP systems.
* The third approach is time-triggered. Once again, the inputs are understood to be continuous signals, and a scheduler determines when a new calculation is performed. The scheduler also makes sure that the calculation "ripples through from left to right" in the correct order. This model is typically used in embedded systems.

Debugging these kinds of systems is relatively straightforward because the calculation is always in a distinct state. Dependencies and data flow, or the currently active block and the available messages, can easily be visualized in a block diagram notation. Note that the calculation rules themselves are considered black boxes here, whose internals may be built from any other paradigm, often functional. Integrating debuggers for the internals of boxes is a more challenging task.

#### 5.2.6 State-based

The state-based paradigm describes a system’s behavior in terms of the states the system can be in, the transitions between these states, events that trigger these transitions and actions that are executed as states change. State machines are useful for systematically organizing the behavior of an entity. They can also be used to describe valid sequences of events, messages or procedure calls. State machines can be used in an event-driven mode in which incoming events actually trigger transitions and the associated actions. Alternatively a state machine can be run in a timed mode, in which a scheduler determines when event queues are checked and processed. Except for possible realtime issues, state machines are easy to debug by highlighting the contents of event queues and the current state[[3]](#footnote-3).

**mbeddr C:** This language provides an extension that supports directly working with state machines. Events can

and simple expression languages, state machines are probably the paradigm that is most often used in DSLs.

be passed into a state machine from regular C code, or by mapping incoming messages in components to events in state machines that reside in components. Actions can contain arbitrary C code, unless the state machine is marked as verifiable, in which case actions may only create outgoing events or change state machine-local variables. J

**Refrigerators:** The behavior of cooling programs is fundamentally state-based. A scheduler is used to execute the state machine at regular intervals. Transitions are triggered either by incoming, queued events or by changing property values of hardware building blocks. This language is an example where a behavioral paradigm is used without significant alterations, but working with domainspecific data structures – refrigerator hardware and its properties. J

State-based behavior description is also interesting in the context of model checking. The model checker either determines that the state chart conforms to a set of specifications or provides a counter-example that violates the specifications. Specifications express something about sequences of states such as "It is not possible that two traffic lights show green at the same time" or "Whenever a pedestrian presses the **request** button, the pedestrian lights eventually will show green"9.

Berard, B., Bidoit, M., Finkel, A.,

In principle, any program can be represented as a state machine and can then be model checked. However, creating state machines from, say, a procedural C program is non-trivial, and the state machines also become very big very quickly. Statebased programs *are already* a state machine, and, they are typically not that big either (after all, they have to be understood by the developer who creates and maintains them). Consequently, many realistically-sized state machines can be model checked efficiently.

### 5.3 Combinations

The behavioral paradigm also plays a role in the context of language composition. If two to-be-composed languages use different behavioral paradigms, the composition can become really challenging. For example, combining a continuous system (which works with continuous streams of data) with a discrete event-based system requires temporal integration. We

|  |
| --- |
| won’t discuss this topic in detail in this book10. However, it . |

is obvious that combining systems that use the same paradigm is much simpler. Alternatively, some paradigms can be integrated relatively easily; for example, it is relatively simple to map a state-based system onto an imperative system.

Many DSLs use combinations of various behavioral and structural paradigms described in this section11. Some combina-

tions are very typical:

* A data flow language often uses a functional, imperative or declarative language to describe the calculation rules that express the dependencies between the variables (the contents of the boxes in data flow diagrams or of cells in spreadsheets). Fig. 4.45 shows an example block diagram, and Fig. 5.5 shows an example implementation.
* State machines use expressions as transition guard conditions, as well as typically an imperative language for expressing the actions that are executed as a state is entered or left, or when a transition is executed. An example can be seen in Fig. 20.7.
* Reactive programming, in which "black boxes" react to events, often using data flow or state-based programming to implement the behavior that determines the reactions.
* In purely structural languages, for example those for expressing components and their dependencies, a functional/expression language is often used to express pre- and post-conditions for operations. A state-based language is often used for protocol state machines, which determines the valid order of incoming events or operation calls.

Note that these combinations can be used to make well-established paradigms domain-specific. For example, in the Yakindu State Chart Tools (Fig. 20.7), a custom DSL can be plugged into an existing, reusable state machine language and editor. One concrete example is an action language that references another DSL that describes UI structures. This allows the state machine to be used to orchestrate the behavior of the UI.

Some of the case studies used as examples in this part of the book also use combinations of several paradigms.

**Pension Plans:** The pension language uses functional abstractions with mathematical symbols for the core actu-

ary mathematics. A functional language with a plain textual syntax is used for the higher-level pension calculation rules. A spreadsheet/data flow language is used for expressing unit tests for pension rules. Various nesting levels of namespaces are used to organize the rules, the most important of which is the pension plan. A plan contains calculation rules as well as test cases for those rules. Pension plans can specialize other plans as a means of expressing variants. Rules in a sub-plan can override rules in the plan from which the sub-plan inherits. Plans can be declared to be abstract, with abstract rules that have to be implemented in sub-plans. Rules are versioned over time, and the actual calculation formula is part of the version. Thus, a pension plan’s behavior can be made to be different for different points in time. J

**Refrigerators:** The cooling behavior description is described as a reactive system. Events are produced by hardware elements[[4]](#footnote-4). A state machine constitutes the top-level struc-

ture. Within it, an imperative language is used. Programs can inherit from another program, overwriting states defined in the base program: new transitions can be added, and the existing transitions can be overridden as a way for an extended program to "plug into" the base program. J

of the hardware element, but in terms of the model the events are associated with the hardware element directly

1. . [↑](#footnote-ref-1)
2. e [↑](#footnote-ref-2)
3. m [↑](#footnote-ref-3)
4. Technically it is of course the driver [↑](#footnote-ref-4)