# Automatic Development of Embedded Systems Using Model Driven Engineering and Compile-Time Virtualisation

Neil Audsley, Ian Gray, Dimitris Kolovos, Nikos Matragkas, Richard Paige and Leandro Soares Indrusiak

**Abstract** The architectures of modern embedded systems tend to be highly application-specific, containing features such as heterogeneous multicore processors, non-uniform memory architectures, custom function accelerators and on-chip networks. Furthermore, these systems are resource-constrained and are often deployed as part of safety-related systems. This necessitates the levels of certification and the use of designs that meet stringent non-functional requirements (such as timing or power). This chapter focusses upon new tools for the generation of software and hardware for modern embedded systems implemented using Java. The approach promotes rapid deployment and design space exploration, and is integrated into a fully model-driven toolflow that supports existing industrial practices. The presented approach allows the automatic deployment of architecture-neutral Java code over complex embedded architectures, with minimal overheads and a run-time support that is amenable to real-time analysis.

## 1 Introduction

Due to their application-specific nature, the architectures of modern embedded systems are commonly very different to that of more general-purpose platforms. Such systems contain non-standard features that are poorly supported by existing languages and development flows, which can make embedded design difficult, slow, and costly.

Good examples of this trend can be observed in recent smartphone devices. The Apple iPhone 3G, released in 2008, contained two main heterogenous processors (an application processor and a baseband processor), four different memory technologies of different speeds and sizes (DDR SDRAM, serial

Department of Computer Science, University of York, UK

flash, NOR flash and SRAM), and a wide range of supplemental processing devices such as touchscreen controllers and power management controllers [10]. In later versions of the device the application processor itself became a heterogeneous, multicore, system-on-chip containing two ARM Cortex-A9 CPUs with a SIMD accelerator, dual core GPU, and dedicated image processing and audio processing cores. Developing software for such a system is extremely challenging and requires large amounts of low-level, hardware-specific software for each part of the system.

The difficulty of software development for complex architectures is compounded by the observation that many embedded systems are deployed in resource-constrained environments and so the efficiency of the final design is a top priority. Also, many embedded systems are real-time systems and so are required to be analysed and certified before deployment to ensure that they are fit for purpose.

This chapter discusses these problems in detail and considers existing solutions in section 2. An approach is then presented that is part of the MADES project, an EU 7th Framework Project [40]. The MADES project uses model-driven techniques to seamlessly integrate model transformation (section 3.2), software generation (section 4) and hardware generation (section 5) flows to promote rapid development, design space exploration, and increase the quality of the final systems. A case study is then presented in section 6 to show how these tool flows work in practice. Finally, the chapter concludes in section 7.

## 2 Background

This section will discuss the unique challenges of embedded development and some of the ways that they are currently addressed. Section 2.1 discusses the complex hardware architectures found in embedded systems, section 2.2 discusses the problems faced by developers of safety-critical and high-integrity systems, and section 2.3 describes industrial concerns.

### *2.1 Heterogenous Hardware Platforms*

The hardware architectures of embedded systems are becoming increasingly non-standard and application specific. Large increases in on-chip transistor density coupled with relatively modest increases in maximum clock rates [21] have forced the exploration of multi-processor architectures with heterogenous processing components in order to meet increasing application performance requirements. Consequentially, many modern embedded systems target Multiprocessor Systems-on-Chip (MPSoCs)-based platforms. These ar-

chitectures are a significant deviation from the homogeneous, uniprocessor platforms that have traditionally been the main component of embedded architectures.

Embedded architectures frequently contain multiple, *heterogeneous* processing elements [25], non-uniform memory structures [3], and non-standard communication facilities (e.g Network-on-Chip communications structures are used on the recent Tilera 64-core TILEPro64 processor [44] and the Intel 48-core Single-Chip Cloud Computer [26]). Embedded systems also make extensive use of application-specific hardware, such as DSP cores, function accelerators, or configurable processors [13]. For example, Texas Instruments' OMAP 5 range of devices [38] contain a dual-core ARM Cortex A15, two other smaller ARM cores, DSPs, and a GPU core.

The lack of a 'standard' architecture means that such systems are not well-supported by the standard toolchains and languages that have been previously developed. This is because the abstraction models of existing programming languages were not developed to cope such variety and variability of heterogeneous platforms. Early computer architectures were largely uniform and entirely static, consisting of a single processor with access to one contiguous block of memory. As a result, the abstraction layers of programming languages hid many architectural details to aid the programmer. This approach has been inherited by modern languages, which increasingly rely on the presence of middleware or a distributed operating system to allow the programmer access to hardware features and architectural mapping. Access to features such as complex memory or custom hardware can only be achieved though the use of abstraction-breaking techniques (link scripts, inline assembly, raw pointers etc.). These techniques are error-prone, difficult to port to new architectures, and hard to maintain. Also, on resource-limited embedded systems complex operating systems or middleware is infeasible.

## *2.2 Criticality*

In addition to the problems described above, embedded systems are frequently deployed in safety-related (i.e. safety-critical) environments, thereby categorising them as hard real-time systems [6]. Such systems must be amenable to worst-case execution time analysis so that their worst-case timing behaviour can be identified and accounted for. This requires predictability at all stages of the design, from language choice (frequently a high-integrity subset such as Ravenscar Ada [5] or Java [24]) through a real-time OS (such as MARTE OS [34]) to real-time hardware features (such as the CAN bus, or SoCBUS [45]).

The heterogenous hardware of embedded systems can often make guaranteeing worst-case timing or resource use very difficult. Many hardware features have highly variable response times. For example, the response time

for a cache is relatively low for a cache hit but very high for a cache miss. Characterising memory accesses as hits or misses at analysis time is an active area of timing analysis research [33, 18], made even harder when multi-level or shared caches are considered.

Once a suitable timing model of the hardware can be constructed that allows analysis, restrictions must be imposed onto the programming model that developers can use in order to support timing analysis of the application software. The commonly used model [6] makes the following assumptions:

- The units of computation in the system are assigned a potentially dynamic priority level.
- At any given time the executing thread can be determined from the priorities in the system and the states of the threads. i.e. Earliest Deadline First scheduling states that the thread with the nearest deadline has the highest priority and should be executing, unless it is blocked.
- Priority inversion (deviations from the above point) in the final system can be prevented, or predicted and bounded.
- Threads contain code with bounded execution times. This implies bounds on loop iterations, predictable paths through functions, restrictions on expected input data, and limitations on exotic language features like code migration, dynamic dispatch, or reflection.
- Blocking throughout the system is bounded and deadlock free.

Finally, once predictable hardware and software are developed it is still necessary for the highest levels of certification (such as the avionics standard DO-178B) to demonstrate traceability from requirements to software elements. Currently this is not well supported by existing toolchains.

### *2.3 Industrial Applicability*

Industry is generally reluctant to switch to new programming languages and toolchains as this imposes a drastically different development approach with implicit problems of risk, acceptance and difficulties with legacy systems. In general, existing industrial methodologies must be supported rather than supplanted. Model-driven engineering (MDE) is becoming more frequently used in industrial projects [29] and represents a common way of tackling the higher abstractions of modern embedded systems [20]. However, as with programming languages it is desirable to remain with existing modelling standards (such as SysML [43] or MARTE [30]) and tooling wherever possible. Another parallel with restricted programming languages is that UML and profiles like MARTE are very complex and there are many different ways to model the same concept, so restricted and more focussed subsets can help with productivity.

### *2.4 Summary*

In summary, the following issues are observed:

- Embedded systems employ complex, heterogeneous, non-standard architectures.
- Such architectures are poorly supported by existing programming methodologies which tend to assume 'standard' hardware architectures.
- Embedded systems are frequently real-time or safety critical systems. This limits the programming model which can be used and the middleware or operating systems that can be deployed.
- Complex embedded architectures are frequently very difficult to analyse for worst-case timing behaviour.
- Industrial developers are reluctant to move to new tools or development methodologies due to concerns over use of legacy code, certification, trust in existing tools, and user familiarity.

## 3 Introduction to Model-Driven Engineering

The approaches introduced in this chapter will leverage Model-Driven Engineering (MDE) to attempt to mitigate some of the problems previously described. This section will introduce MDE, metamodels, and model transformations, and then describe the model transformation framework that is used throughout the work described by this chapter.

MDE is a software development paradigm, which aims to raise the level of abstraction in system specification and to increase the level of automation in system development. In MDE, models, which describe different aspects of the system at different levels of abstraction, are promoted to primary artifacts. As such, models "drive" the development process by being subjected to subsequent transformations until they reach a final state, where they are made executable, either by code generation or model interpretation.

MDE relies on two facts [22]. First, any kind of system can be represented by models and second, any model can be automatically processed by a set of operators. Since, models need to be understood and processed by machines, they need to conform to a metamodel. Metamodels are used as a typing system to provide precise semantics to the set of models they describe. Therefore, a metamodel is a model, which defines in a precise and unambiguous way a class of valid models. The metamodel describes the abstract syntax of a modelling language. The homogeneity of definition provided by metamodels enables engineers to apply operations on them such as transformations or comparisons in an automatic and generic way. Figure 1 illustrates the basic relations of conformance and representation between a system, a model and its corresponding metamodel.

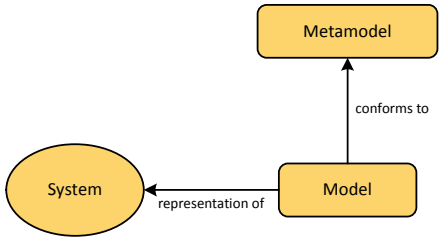


**Fig. 1** Basic relations of representation and conformance in MDE (adapted from [22]).

### *3.1 Model Transformations*

Model transformations play a key role in model-driven development. Czarnecki and Helsen [8] identify the following areas in which they are most applicable:

- Generating lower-level models and code from higher-level, more abstract models;
- Mapping between different models;
- Querying and extracting information from models;
- Refactoring models;
- Reverse engineering of abstract models from concrete ones.

Model transformations are computer programs, which define how one or more input models can be transformed into one or more output models. A model transformation is usually specified as a set of relations that must hold for a transformation to be successful. The input and output models of the transformation have to conform to a metamodel.

A model transformation is specified at the metamodel level and establishes a mapping between all the models, which conform to the input and output metamodels. Model transformations in MDE follow the model transformation pattern illustrated in figure 2. The execution of the rules of a transformation program results in the automatic creation of the target model from the source model. The transformation rules, as well as the source and target models conform to their corresponding metamodels. The transformation rules conform to the metamodel of the transformation language (i.e. its abstract syntax), the source model conforms to the source metamodel and the target model conforms to the target metamodel. At the top level of this layered architecture lies the meta-metamodel, to which all the other metamodels conform.

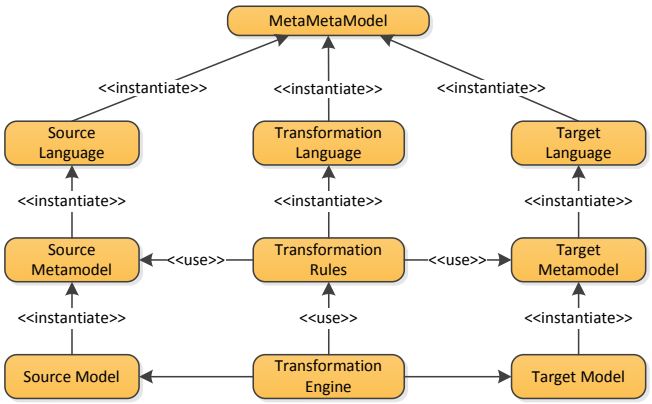


**Fig. 2** Model transformation pattern [4].

## *3.2 Epsilon Model Transformations*

Model transformation languages are used to specify model transformations. In general, model transformations may be implemented in different ways, for example, by using a general purpose programming language or by using dedicated, domain specific model management languages.

In the context of MADES, the model transformation language used is the Epsilon Generation Language (EGL) [35], which is the model-to-text transformation language of the Epsilon framework [9]. Epsilon (Extensible Platform of Integrated Languages for mOdel maNagement) is a family of consistent and interoperable, task-specific, programming languages which can be used to interact with models to perform common MDE tasks such as code generation, model-to-model transformation, model validation, comparison, migration, merging and refactoring.

Epsilon consolidates the common features of the various task-specific modelling languages in one base language and then develops the various model management languages atop it. The Epsilon Connectivity Layer (EMC) abstracts different modelling frameworks and enables the Epsilon task-specific languages to uniformly manage models of those frameworks. The architecture of the Epsilon framework is illustrated in figure 3.

The approach proposed by this chapter is not dependent on the model management framework. However, Epsilon was preferred because of some of its unique features simplify the implementation activities. Such features include the support of Epsilon for interactive model transformations, the fine-grained traceability mechanism of EGL, as well as the framework's focus on reusability and modularity. Moreover, Epsilon is a mature model management framework with an active and large community.

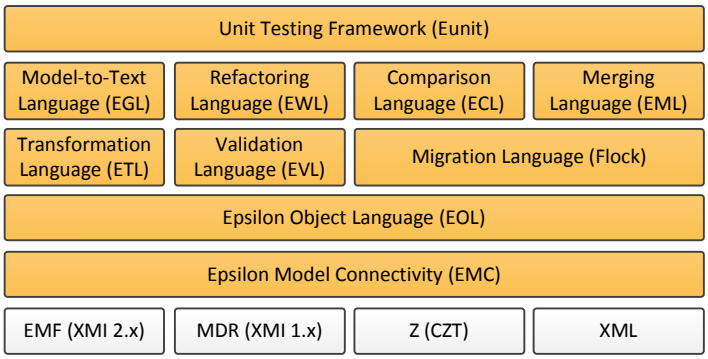


**Fig. 3** Epsilon Framework Architecture.

## 4 Software Generation Using Compile-Time Virtualisation

Given the problems highlighted in section 2, it can be seen that software development for many modern embedded systems is very challenging. Any solution to these problems must be industrially-acceptable so from the discussions in sections 2.3 and 2.2 the following requirements can be obtained:

- No new programming languages or tools because of certification requirements.
- No large runtime layers, or complex translated code.
- Integration with model-driven development to aid developers.

The MADES project therefore uses a model-driven approach which integrates a technique called Compile-Time Virtualisation (CTV) [15, 16]. Section 4.1 describes CTV and motivates its use while section 4.2.3 describes how CTV is integrated into MADES.

### *4.1 Compile Time Virtualisation (CTV)*

Compile Time Virtualisation (CTV) is a source-to-source translation technique that aims to greatly simplify the development of software for embedded hardware architectures. It does this by integrating hardware virtualisation to hide the complexities of the underlying embedded architecture in a unique way that imposes minimal runtime overheads and is suitable for use in real-time environments. CTV allows the developer to write software for execution on a 'standard' desktop-style environment without having to consider the target platform. This architecturally-neutral input software is automatically translated to architecturally-specific output software that will

execute correctly on the target hardware. The output software is supported by an automatically-generated, minimal-overhead, runtime that avoids the code size increase of standard middleware technologies (such as CORBA [32]) and run-time virtualisation-based systems (such as Java). CTV is a language-independent technique that can be applied to a range of source languages. It has currently been demonstrated in C [15] and Java [17]. The rest of this chapter will discuss CTV as it is applied to Java, but the approach is broadly the same in all languages.

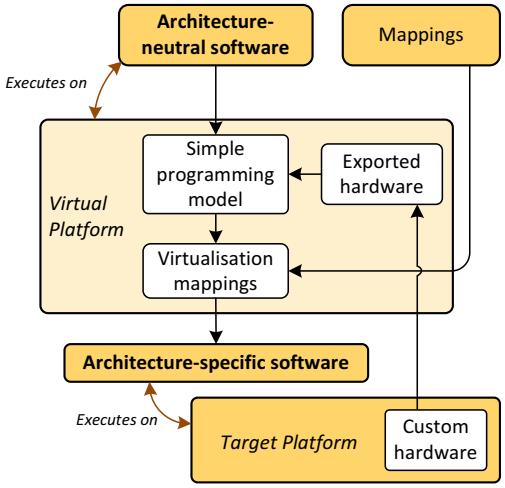


**Fig. 4** Compile-Time Virtualisation introduces a Virtual Platform to make software development easier.

CTV introduces a virtualisation layer over the target hardware, called the *Virtual Platform* (VP). This is shown in figure 4. The VP is a high-level view of the underlying hardware that presents the same programming model as the source language (in this case Java) to simplify development. For Java, it presents a homogeneous symmetric multiprocessing environment with a single monolithic shared memory, coherent caching, and a single uniform operating system. This is equivalent to a standard desktop computer running an operating system like Linux or Windows and is the environment in which Java's runtime expects to operate. *Therefore, the developer can write normal, architecture-independent Java code.*

As its name implies, the VP is a *compile-time only* construct, it does not exist at run-time. This is because the VP's virtualisation is implemented by a source-to-source translation layer that is guided by the virtualisation mappings (that map threads to CPUs and data to memory spaces). This can be seen in figure 5. The job of the source-to-source translation is to translate the architecturally-independent input software into architecturally-specific

output code that will operate correctly on the target hardware, according to the provided mappings.

Unlike a standard run-time virtual machine, the virtualisation mappings are exposed to the programmer. This allows the programmer to influence the implementation of the code and achieve a better mapping onto the architecture. For example, by placing communicating threads on CPUs that are physically close to each other, or locating global data in appropriate memory spaces to minimise copying. Such design space exploration can be performed very rapidly because software can be moved throughout the target system without recoding.

Also in contrast to run-time virtual machines, custom hardware can be exported up to the programmer through the VP at design-time and presented in a form that is consistent with the source language's programming model, thereby allowing it to be effectively exploited by the programmer.

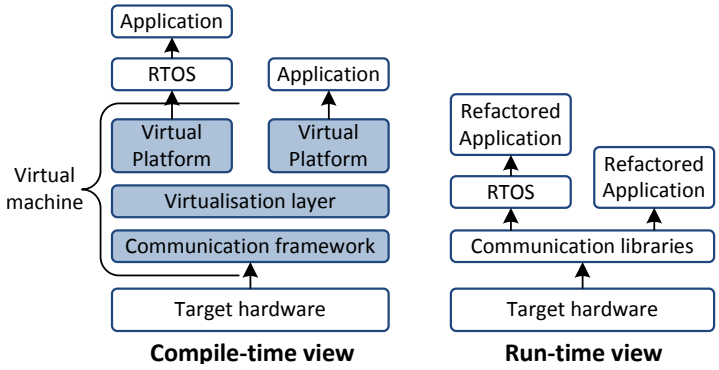


**Fig. 5** Compile-Time Virtualisation hides complex hardware, but only at compile-time.

Moving the virtualisation to compile-time rather than run-time helps to reduce run-time overheads. Such overheads in a CTV system are small because all the work is done by the refactoring engine at compile-time. However, a consequence of applying the refactoring at compile-time is that all necessary analysis must be able to be be performed offline. This means that certain aspects of the input language are restricted. However, as discussed in section 2.2, in a real-time system such restrictions are already imposed (e.g. in the Ravenscar [5, 24] and SPARK [19] real-time language subsets). For more detail on this, see section 4.2.2. In general, the principle is that:

> *A system which is implemented using Compile-Time Virtualisation trades runtime flexibility for predictability and vastly reduced overheads.*

For examples of how this trade off can reduce overheads, see section 4.2.4.

Some additional benefits of the VP is that its use abstracts hardware changes from the software developer. The developer only has to target the

VP rather than the actual hardware and if the hardware is changed at a later date, the same software can be retargeted without any recoding or porting effort. Similarly, because the VP is implemented to support development in existing languages, developers do not have to be trained to use a new language and existing legacy code can be more easily reused. Also, because the architecture-specific output code is still valid Java, no new compilers or tool need to be written. This is of vital importance to high-integrity systems that require the use of trusted compilers, linkers, and other tools.

The CTV approach is different to techniques such as Ptolemy II [11] which aim to provide new higher-level and more appropriate abstractions for programming complex systems. CTV is instead designed to allow existing languages and legacy code to be used to effectively target such systems through the use of very low-overhead virtualisation. The two different approaches can actually be complementary and used together, with CTV used as a low-overhead intermediary to bring legacy code or legacy programming languages into an otherwise Ptolemy-defined system.

CTV is the name for the general technique. Section 4.2 will now discuss AnvilJ, the specific implementation of CTV that is implemented in the MADES project.

## *4.2 AnvilJ*

Section 4.1 gave a broad overview of CTV. However, CTV is a language-independent technique that can be implemented to work with a range of input languages. In the MADES project the chosen source development language is Java (and its real-time variants [14, 24]), therefore MADES uses *AnvilJ*, a Java-based implementation of CTV that is described in the rest of this section. The AnvilJ system model is described in section 4.2.1. Whilst AnvilJ can accept the majority of standard Java, a few restrictions must be imposed and these are enumerated in section 4.2.2. The way that MADES integrates AnvilJ into its model-based design flow is discussed in section 4.2.3, and section 4.2.4 concludes with a discussion of how AnvilJ results in a system which displays minimal runtime overheads.

### 4.2.1 AnvilJ System Model

AnvilJ is an implementation of CTV for the Java programming language and its related subsets aimed at ensuring system predictability, such as the RTSJ. The AnvilJ system model is shown in figure 6. Its input is a single Java application modelled as containing two sets:

- **AnvilJ Threads:** A set of `static final` instances or descendants of `java.lang.Thread`.

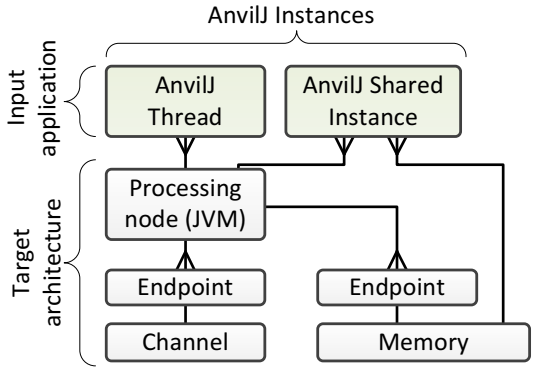


**Fig. 6** The AnvilJ system model.

- **AnvilJ Shared Instances:** A set of `static final` instances of any other class.

Collectively, AnvilJ Threads and Shared Instances are described using the umbrella term *AnvilJ Instances*. AnvilJ Instances are static throughout the lifetime of the system; they are created when the system starts and last until system shutdown.

An AnvilJ Instance may communicate with any other AnvilJ Instance, however the elements it has created may not communicate with the created elements of other AnvilJ Instances. This restriction allows the communication topology of the system to be determined at compile-time and the required runtime support to be reduced, as discussed later. This approach is particularly suited to embedded development because it mirrors many of the restrictions enforced by high-integrity and certification-focussed language subsets (such as the Ravenscar subsets of Ada [5] and Java [24] or the MISRA-C coding guidelines [41]).

In AnvilJ, the main unit of computation in the target hardware is the **processing node**. A processing node models a Real-Time Java Virtual Machine (JVM) [31] in the final system (or a standard JVM with accordingly reduced predictability). The Java specification does not define whether a multicore system should contain a single JVM for the entire system [37, 1] or one per core. Therefore AnvilJ models the JVMs, rather than the processors. The JVMs need not have similar performance characteristics or features. As with all CTV implementations, every AnvilJ Instance is mapped to exactly one node. AnvilJ Instances cannot migrate between processing nodes, but (if supported by the Java implementation) other instances can.

Nodes communicate using **channels**, which are the communication primitives of the target architecture. AnvilJ statically routes messages across the nodes of the system to present the totally-connected communications assumed by Java. The designer provides drivers for the channels of the system. **Memories** represent a contiguous logical address space and **endpoints** connect

processing nodes to other hardware elements. Every AnvilJ Shared Instance must be mapped to either exactly one node (on the heap of the JVM), or exactly one memory where it will be available to all nodes connected to that memory.

This model is compile-time static – the number of AnvilJ Instances does not change at runtime. This is consistent with the standard restrictions that are imposed by most real-time programming models (as discussed in section 2.2. For example, Ravenscar Ada [5] forbids all dynamic task allocation, whereas AnvilJ only forbids dynamic AnvilJ Instances. This is in contrast to systems like CORBA which adopt a "dynamic-default" approach in which runtime behaviour is limited only by the supported language features. Such systems support a rich runtime model but the resulting system can be heavyweight as they are forced to support features such as system-wide cache coherency, thread creation and migration or dynamic message routing, even if not required by the actual application. The approach of CTV is "static-default" in which the part of the application modelled is static. The restricted programming model promises less, but the amount of statically-available mapping information allows the required runtime support to be significantly reduced.

Not all instances of `java.lang.Thread` need to be modelled as an AnvilJ Instance. Equally, not all shared object instances need to be modelled at all. Enough should be modelled to fulfill the constraint that program instances created by an AnvilJ Thread $t$ only communicate with other instances created by $t$, or another AnvilJ Instance.

### 4.2.2 Restrictions on Input Code

In order to be correctly refactored, AnvilJ input programs must be written to conform to a small set of restrictions which are detailed in this section. These restrictions are consistent with those required by existing real-time development processes (i.e. SPARK [19] or MISRA-C [41], see section 2.2) and in most cases are less restrictive. They allow the system to operate with hugely reduced runtime overheads (see section 4.2.4).

- AnvilJ threads and shared objects must be declared as `static final` fields. This means that the refactoring engine can determine at compile-time their location and number, which is not in general possible otherwise.
- All accesses to an AnvilJ object must directly refer to the field (using dot notation if the reference is in another class). It is forbidden to 'leak' a reference to an AnvilJ object, for example by returning it from a method, passing it to a method, or assigning it to a local variable of another class. Any of these actions will be checked by the refactoring engine and prevented.

- The arguments and return values of shared methods that are exported by an AnvilJ thread or shared object must implement `java.io.Serializable` interface.
- Threads on different nodes must only use other AnvilJ objects to communicate. Threads may perform any action that only affects the local JVM. However, if it calls methods or accesses fields with an instance on a different node that instance must be tagged as an AnvilJ Instance.

#### 4.2.3 Integration With Model-Driven Engineering

To aid the use of AnvilJ, MADES integrates it directly into the model-driven engineering (MDE) flow of the project. This is not mandatory for AnvilJ, which can be used independently. In order to integrate AnvilJ it is necessary to provide the designer with a way of expressing a high-level view of the target hardware (in terms of the AnvilJ system model) and a high-level view of relevant parts of the input software. Not all the input software needs to be modelled, only the parts that are to be marked as AnvilJ Instances (section 4.2.1). Also, the allocation of AnvilJ instances from the software model to the processing nodes of the hardware model must be provided.

This information is then translated from the designer's model into the form which is required by the AnvilJ tool. The translation is implemented using the Epsilon model transformation language, which is described in detail in section 3.2. In the MADES framework, this information is provided by the designer through the use of 13 stereotypes which are applied to classes in the system model. These MADES stereotypes are described in table 1. The modelling tool used in the MADES flow (Modelio [28]) supports two additional diagram types that use the MADES stereotypes; the detailed hardware specification and the detailed software specification. Allocations are performed with a standard allocation diagram. Working with these additional diagrams aids the designer because the MADES stereotypes can be automatically applied.

For a more detailed look at how the modelling is performed to integrate AnvilJ, section 6 presents a case study that shows the development of a subcomponent of an automotive safety system.

#### 4.2.4 Overheads

AnvilJ's static system model allows most of the required support to be implemented at compile-time, resulting in a small runtime support system, especially when compared with much larger (although more powerful) general-purpose frameworks. As will be shown in this section, the main overhead in an AnvilJ system is that of the Object Manager (OM). The OM is a microkernel which exists on every processing node of the system and implements

**Table 1** Brief description of the MADES stereotypes

| Stereotype | Description |
|---|---|
| «mades_hardwareobject» | Superstereotype for all hardware stereotypes. |
| «mades_clock» | Connected to «mades_processingnode» instances and «mades_channel» instances to denote a logical clock domain. |
| «mades_channel» | A communication resource i.e. bus |
| «mades_ipcore» | Additional hardware i.e function accelerator. |
| «mades_memory» | A single logical memory device. |
| «mades_processingnode» | A computation element of the hardware platform. Commonly this is a single processor, but as described in section 4.2.1, this corresponds to a JVM in the final system. |
| «mades_endpoint» | Superstereotype of all endpoint stereotypes. Endpoints connect processing nodes to other hardware and provide more information about the connection. i.e. an ethernet endpoint may provide a MAC address. |
| «mades_memorymedia» | Connects a «mades_processingnode» instance to a «mades_memory» instance. |
| «mades_devicemedia» | Connects a «mades_processingnode» instance to a «mades_ipcore» instance. |
| «mades_channelmedia» | Connects a «mades_processingnode» instance to a «mades_channel» instance. |
| «mades_softwareobject» | Superstereotype for all software stereotypes. |
| «mades_thread» | Represents an AnvilJ Thread. |
| «mades_sharedobject» | Represents an AnvilJ Shared Instance. |

the AnvilJ system model. The OMs use a message-passing communications model to implement shared memory, locks, remote method calls etc.

The full version of the OM compiles to approximately 34kB of class files including debugging and error information. However it is also possible to create smaller OMs which only support a subset of features for when the software mapped to a node does not require them. For example, if a node contains AnvilJ Shared Instances but no AnvilJ Threads then 5.7kB of support for 'Thread creation and joining' can be removed. If none of the shared methods of a node are called then the shared methods subsection can be removed. The advantage of AnvilJ's offline analysis is that this can be done automatically each time, based on the exact input application and hardware mappings. Figure 2 shows a breakdown of some of the feature sets of the OM and their respective code footprint.

Figure 7 compares this size to other similar systems. It should be noted that this comparison is provided purely to contextualise the size metric and demonstrate that AnvilJ's size is small, relative to related embedded frameworks. The other systems graphed, especially the CORBA ORBs, are built to support general-purpose, unseen software and consequentially are much more heavyweight.

**Table 2** Class file sizes for OM features

| Feature set | Approx. size |
|---|---|
| Thread creation and joining | 5.7 kB |
| Remote Object Locks | 4.5 kB |
| Shared methods | 8.4 kB |
| Sockets-based IComms (debug) | 4.29 kB |
| Full OM | 34 kB |

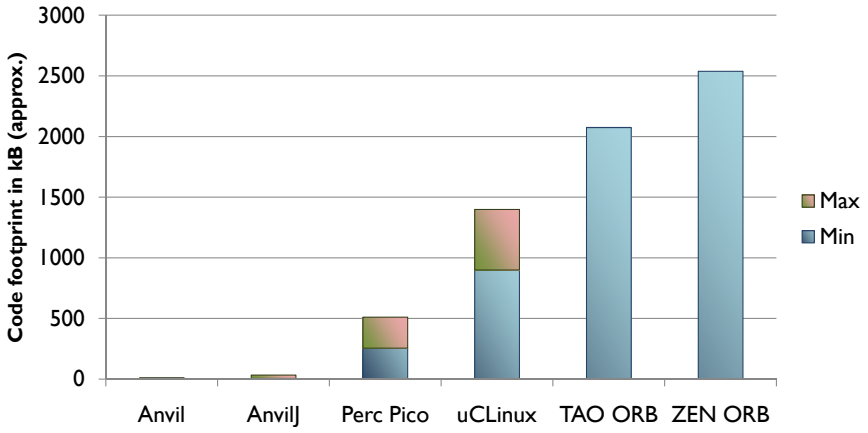


**Fig. 7** The code footprint of the AnvilJ runtime compared to systems from a similar domains. Anvil is a C-based implementation of CTV, Perc Pico [2] implements safety-critical Java on systems without an OS, uClinux is a reduced Linux kernel for microprocessors with MMUs, and TAO [36] and ZEN [23] are Real-Time CORBA ORBs.

In addition to the small code size of the OM, its runtime memory footprint is also modest. The full OM in a desktop Linux-based system uses approximately 648 bytes of storage when idle, which increases as clients begin to use its features.

## 5 Hardware Generation Using Model Transformations

The MADES hardware generation flow transforms a detailed hardware specification diagram into an implementable hardware description. The generated hardware may be a complex, heterogeneous system with a non-uniform memory architecture but it is supported and programmed by the software generated by the code generation transformations described in section 4.

In order to best demonstrate the flexibility of the hardware generation flow, the translations target Xilinx FPGAs. This is merely an implementa-

tion choice and does not reflect any part of the flow which inherently requires Xilinx devices and tools (or FPGAs in general). Other implementation structures can also be supported. The transformation outputs a Microprocessor Hardware Specification (MHS) file [47] which is passed to Xilinx Platgen, a tool that is part of Xilinx's Embedded Development Kit design tools [48]. Platgen is a tool which reads an MHS file and outputs VHDL [7] which can then implemented on the target FPGA. This flow is illustrated in figure 8.

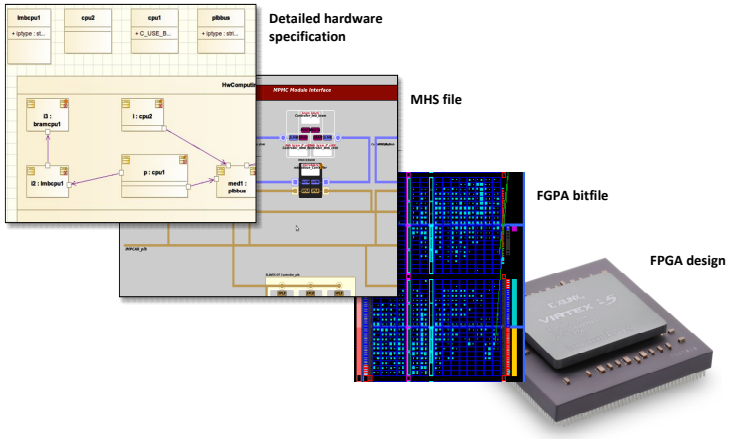


**Fig. 8** The hardware generation flow.

The hardware generation flow is implemented using the Epsilon Generation Language (EGL) (see section 3.2). There are three main benefits gained from generating hardware from the system model in this way:

- Very rapid prototyping and design space exploration can be achieved using this method due to the fact that hardware architectures can be constructed in the developer's modelling environment rather than vendor tools.
- MDE allows a vendor-neutral way of modelling and generating architectures. The same models could be used to target a wide range of FPGAs, ASICs, or even other hardware description languages like SystemC, however such an approach would not support the full flexibility of these systems.
- The same model is used as a source for both the software generation and hardware generation flows. These models share a consistent meta-model and so have related semantics. This gives confidence in the final design, because the software generation flow is refactoring code according to the same hardware model used by the hardware generation flow. In essence, the two flows 'meet in the middle' and support each other.

When creating the detailed hardware specification diagram, the hardware only needs to be modelled at a high level of abstraction. The platform is

modelled as a class stereotyped with the stereotype «mades_architecture». Each detailed hardware specification contains exactly one such class. Properties in the «mades_architecture» stereotype are used to guide the software generation process by denoting the entry point class of the input application and allocating the initial `Main` thread to a processing node.

The details of the architecture are modelled with the MADES hardware stereotypes. Processing nodes ( «mades_processingnode») are the elements of computation in the platform. Each node supports a logical JVM. They communicate with other nodes through the use of channels. Nodes connect to channels using the «mades_channelmedia» endpoint stereotype. Memories ( «mades_memory») are data-storage elements and are connected to channels using «mades_memorymedia». Other hardware elements ( «mades_ipcore») are connected to channels through the use of the «mades_devicemedia» endpoint stereotype.

The top-level hardware stereotype «mades_hardwareobject» defines a property called `iptype`. This is passed to the hardware generation transformation to specify the type of hardware which should be instantiated. Further properties can also be passed depending on the value of `iptype`. For an example of this see the case study in section 6.5.

Clock domains are modelled by classes stereotyped with the «mades_clock» stereotype. Clock synthesis is restricted by the capabilities of the implementation target and the IP cores used. A set of design rules are first checked using model verification to ensure that the design can be realised. These are:

- The total number of clock domains is not higher than the limit for the target FPGA.
- All communications across clock boundaries use an IP core that is capable of asynchronous signalling (such as a mailbox).
- All IP cores that require a clock are assigned one.

Each clock has a target frequency in the model and is implemented using the clock manager cores of the target FPGA. As with all FPGA design, the described constraints are necessary but not sufficient conditions. During synthesis the design may use more clock routing resources than are available on the device, in which case the designer will have to use a more powerful FPGA or reduce the clock complexity of the design.

Currently, interfaces (IO with the outside world) have to be taken from the IP library or manually defined in VHDL or Verilog. It is not the aim of this approach to provide high-level synthesis of hardware description languages such as in Catapult-C [27] or Spec-C [12], although such approaches can be integrated by wrapping the generated core as an IP core for the Xilinx tools.

## 6 Case Study: Image Processing Subsystem

This section will present a case study to illustrate the benefits of the MADES Code Generation approach and show how CTV/AnvilJ is integrated into the design flow. This case study will detail the development of a subsection of an automotive safety system called the Car Collision Avoidance System (CCAS). The CCAS detects obstacles in front of the vehicle to which it is mounted and, if an imminent collision is detected, applies the brakes to slow the vehicle. In this case study we focus on a small part of the detection subsystem and show how the MADES code generation allows architecture-independent software to be generated to process the radar images without concern for the target platform. Multiple hardware architectures can be modelled and the software automatically deployed over auto-generated hardware.

Section 6.1 gives a block-level overview of the developed component and section 6.2 discusses how the initial software is developed. The generation of the software and hardware models is covered in sections 6.3 and 6.4. The generation of the target hardware is detailed in section 6.5 and finally section 6.6 discusses deploying the software to the generated hardware.

### *6.1 Subsystem Overview*

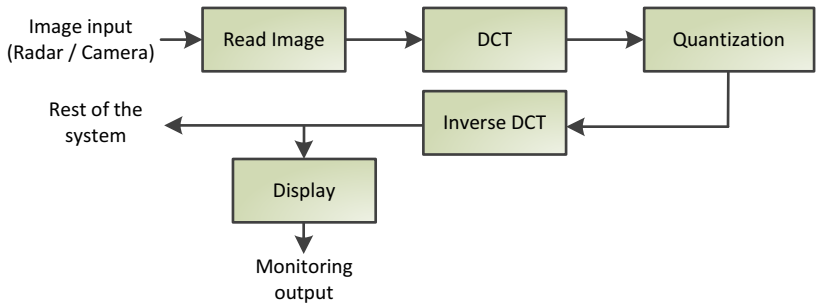


**Fig. 9** Block diagram of the implemented subsystem. A monitoring output stage is included to allow verification of the subsystem during system development.

The developed subsystem takes images from the radar (or camera) and applies JPEG-style compression to reduce the size of the image and therefore reduce the demand on on-chip communications. Once reduced in size, the images are passed on to other parts of the system for feature extraction and similar algorithms. The block diagram of the subsystem is given in figure 9. The main stages of the subsystem are as follows:

- **Read Image:** Periodically reads images from the input to the system from a radar or camera.
- **DCT:** A Discrete Cosine Transformation moves the representation of the image from the spatial domain into the frequency domain.
- **Quantization:** Data in the frequency domain is selectively discarded to compress the image.
- **Inverse DCT:** Moves the image back into the spatial domain. The result is a (compressed) image that can be passed to the rest of the system, or optionally fed to a monitoring output stage.
- **Display:** Used for monitoring and debugging, the output stage uses a graphical user interface to display the image to the user.

### *6.2 Software Development*

Developing the software for this subsystem is very simple when using AnvilJ because the developer can develop as if the code will execute on a standard desktop Java environment. However, the developer must observe the restrictions detailed in section 4.2.2. Also it is not possible to develop the low-level drivers for the radar/camera input through AnvilJ directly, so for the purpose of testing and initial development stub drivers should be used that operate on the development platform. Final hardware interfacing must be done once deployment is underway as is normal practice.

The main restriction imposed by AnvilJ is that AnvilJ Instances must be static and only communicate through other AnvilJ Instances. This forces the developer to consider the structure of their code carefully, as is the case with all embedded development. The refactoring engine of AnvilJ allows the entire operation of the subsystem to be detailed using a single Java program, even though the final hardware platform may involve multiple heterogeneous processing elements. The code was structured as follows:

- Each block of the subsystem (see figure 9) is implemented as a `static final` thread. The threads are declared and started by a `Main` class that is responsible for initialising the system.
- Each thread contains internal state that holds images passed into it from the previous stage, and methods that allow the previous stage to pass in images to process. The thread processes images in its work queue, and passes completed images to the next thread.
- Each thread is designated as an AnvilJ Thread. This ensures that all communications in the system go through AnvilJ Instances.
- The output stage is designated an AnvilJ Shared Instance.
- Standard implementations of the DCT and Quantize stages are used from open source, freely-available code. This is one of the great advantages of AnvilJ in that often legacy code can be integrated easily.

Having created the software, its functionality can be verified immediately simply by executing the code in the development environment. It is not necessary to use simulators, cross-compilation or similar. The result of the software operating on a test image is shown in figure 10 and a listing of the `Main` class can be found in figure 11. Note that the listing is standard Java code, no extra-linguistic features are required.

**Fig. 10** Example of the architecturally-neutral software operating in the development environment on a test image. The right-hand image is after processing.

## *6.3 Software Modelling*

In a model-driven development flow, the architecture-independent software will be developed based on a software model. This model must be extended with a MADES 'Detailed Software Specification' diagram (detailed in the previous chapter) to inform the AnvilJ tool of the AnvilJ Instances that are present in the input software. This diagram links elements of the software model with the input software, using the concept of 'bindings'.

Bindings are a way of uniquely identifying source code elements (classes, instances, fields, methods etc.) and are defined by the Eclipse JDT project [39]. The developer obtains the binding for an AnvilJ Instance from the AnvilJ GUI and adds it to the `binding` property of the «mades_softwareobject» stereotype in the software model. This links the instance in the detailed software specification diagram to the source code.

Figure 12 shows the completed detailed software specification diagram. The diagram is very simple as its only purpose is to add AnvilJ Elements to the software model and link them to the source code with binding keys. Note that the use of the «mades_thread» and «mades_sharedobject» stereotypes.

```
public class Main {
  private final static int QUALITY = 20;

  public static final ReadThread readThread =
    new ReadThread();
  public static final DCTThread dctThread =
    new DCTThread(QUALITY);
  public static final QuantizeThread quantizeThread =
    new QuantizeThread(QUALITY);
  public static final OutputStage outputStage =
    new OutputStage(QUALITY);

  public static void main(String[] args) {
    readThread.start();
    dctThread.start();
    quantizeThread.start();

    readThread.join();
    dctThread.join();
    quantizeThread.join();
  }
}
```

**Fig. 11** Listing of the `Main` class that initialises the implemented subsystem.

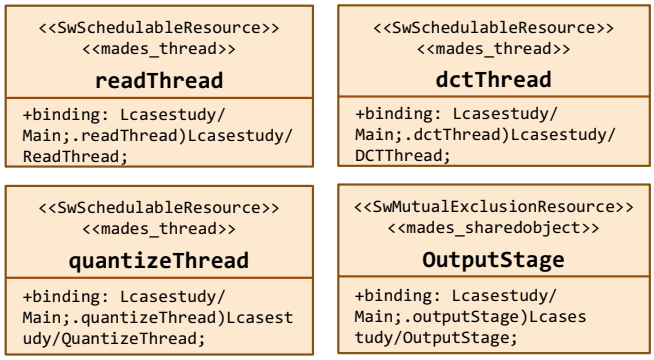


**Fig. 12** The detailed software specification diagram for the case study subsystem.

## *6.4 Hardware Modelling*

Having modelled the software, this section will now describe how the target hardware platform is modelled for AnvilJ integration. Recall that according to the AnvilJ system model from section 4.2.1, it is only necessary for the hardware model to cover a high-level view of the capabilities of the target platform; in terms of processing nodes, memories, channels, and application-specific IP cores.

In this case study we will describe two target platforms and show how the same input software can be automatically deployed without recoding. The first presented architecture is a dual-processor system with a non-uniform memory architecture, shown in figure 13.

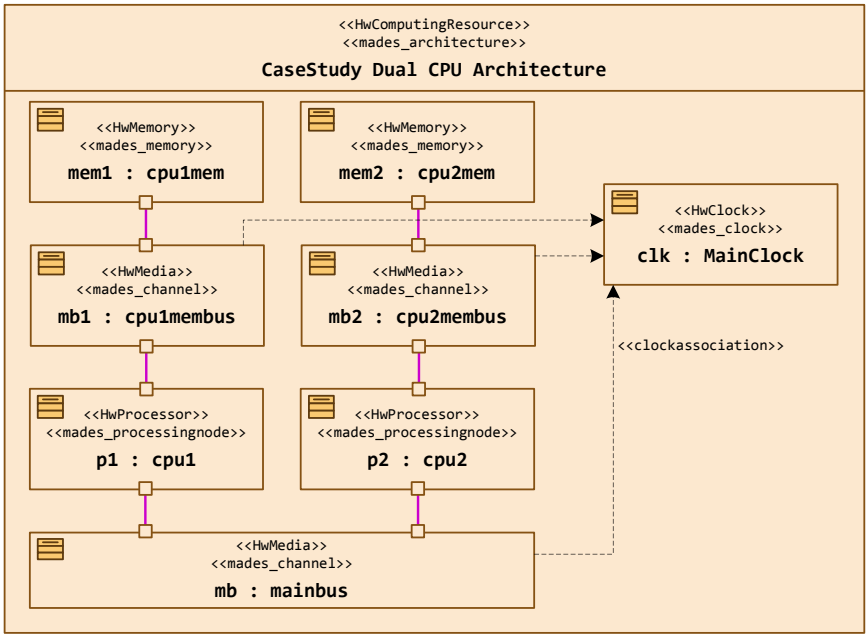


**Fig. 13** The detailed hardware specification diagram for the case study target architecture. Not shown are properties in the classes that describe each hardware element in greater detail.

Once the detailed hardware model is complete, the hardware generation flow can be initiated.

## *6.5 Hardware Generation*

The designer uses the MADES model transformations of section 3.2 to transform the architecture modelled in section 6.4 into an implementable hardware description. As discussed previously in section 5, the hardware generation flow targets Xilinx FPGAs and uses the Xilinx IP libraries from Xilinx Embedded Development Kit [48]. Accordingly, the hardware model must be augmented to include enough details to instantiate these IP cores. This is done by adding properties to the classes of the detailed hardware specification diagram. Full

details of these properties are outside the scope of this chapter and are given in the MADES documentation [40].

Each of the MADES hardware stereotypes has a mandatory property called `iptype`. This is used by the Epsilon model transformation to inform it which Xilinx IP should be instantiated. Each supported IP has a set of *attributes* that may be also set from the model. For example, the `xps_uartlite` IP core is a serial transceiver and includes attributes such as `C_BAUDRATE` to set the expected baud rate and `C_USE_PARITY` to switch on or off the use of parity bits. The hardware generation flow checks for the presence of any mandatory attributes and warns the user if they are not present.

```
PORT fpga_0_uart_RX_pin = fpga_0_uart_RX_pin, DIR = I
PORT fpga_0_uart_TX_pin = fpga_0_uart_TX_pin, DIR = O
PORT fpga_0_mainClk_pin = clock_mainClk, DIR = I,
  SIGIS = CLK
PORT fpga_0_sys_1_rst_pin = sys_rst_s, DIR = I,
  SIGIS = RST, RST_POLARITY = 1

BEGIN microblaze
 PARAMETER INSTANCE = cpu1
 PARAMETER C_USE_BARREL = 1
 PARAMETER HW_VER = 8.20.b
 PARAMETER C_DEBUG_ENABLED = 0
 BUS_INTERFACE DPLB = plbbus
 BUS_INTERFACE IPLB = plbbus
 PORT MB_RESET = mb_reset
 BUS_INTERFACE ILMB = cpu1_ilmb
 BUS_INTERFACE DLMB = cpu1_dlmb
END
```

**Fig. 14** Fragment of the MHS generated by transforming the case study architecture of figure 13.

Once the model is completed with the required information, the user runs the hardware transformation to produce a Xilinx MHS file. A fragment of the MHS generated by transforming the case study architecture of figure 13 is shown in figure 14. This MHS file must be then converted into VHDL using the Xilinx design tools. For the purpose of this case study, the target will be a Xilinx Virtex 5 FPGA [46]. At the end of the implementation, an FPGA bitfile will be created which can then be programmed to the device for testing.

## *6.6 Code Deployment*

After modelling the software and hardware, a deployment diagram can be created that maps instances from the detailed software specification to the detailed hardware specification. For this case study, the initial allocation will locate the image reading thread to CPU1 and all other threads to CPU2. The diagram that performs this allocation can be seen in figure 13.

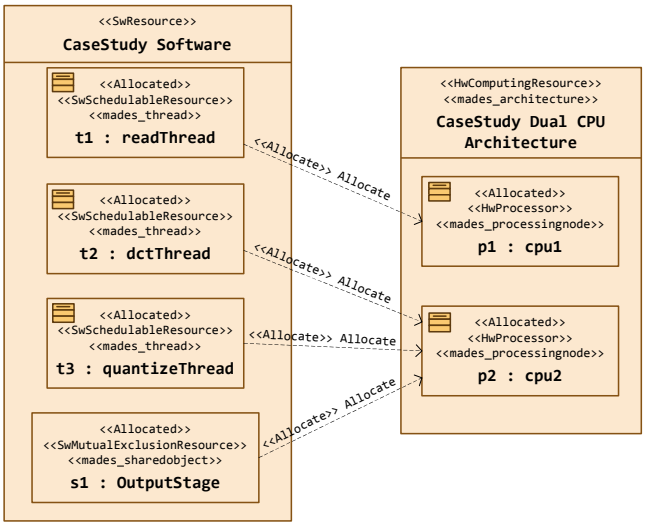


**Fig. 15** An allocation diagram that deploys software from the detailed software specification diagram of figure 12 to the detailed hardware specification of figure 13.

With the addition of the allocation diagram the is model is now complete, so it is exported in XMI format for use in the Eclipse IDE. Once imported to Eclipse, an Epsilon model transformation is used to create an AnvilJ architecture description. This file is created from the hardware, software, and allocation diagrams and is the input to the AnvilJ refactoring engine. It tells AnvilJ what the structure of the input software will be, which elements are AnvilJ Instances, the topology of the target platform, and how to place the AnvilJ Instances throughout the platform. Figure 16 shows the architecture description for the case study.

Once an architecture description is created, the AnvilJ refactoring engine can be invoked at any time to refactor the architecturally-neutral Java application (an Eclipse project) into a set of architecturally-specific output programs, one for each processing node of the target platform as described in the hardware diagram. As the case study architecture has two processing nodes, two output projects will be created. AnvilJ is fully-integrated into the Eclipse Development Environment. After refactoring is complete, the output

```
<architecture name="CaseStudy Dual CPU Architecture"
    mainclass="casestudy.Main" maincpuid="0">
  <cpu name="cpu1" id="0">
    <thread binding="Lcasestudy/Main;.readThread)
      Lcasestudy/ReadThread;"/>
  </cpu>
  <cpu name="cpu2" id="1">
    <thread binding="Lcasestudy/Main;.quantizeThread)
      Lcasestudy/QuantizeThread;"/>
    <sharedobject binding="Lcasestudy/Main;.outputStage)
      Lcasestudy/OutputStage;"/>
    <thread binding="Lcasestudy/Main;.dctThread)
      Lcasestudy/DCTThread;"/>
  </cpu>
  <channel name="plbbus">
    <endpoint cpu="cpu1"/>
    <endpoint cpu="cpu2"/>
  </channel>
</architecture>
```

**Fig. 16** The AnvilJ architecture description for the case study. Note the binding keys correlate with those of the software diagram in figure 12.

applications can be verified by executing both. AnvilJ's default implementation uses TCP sockets for inter-node communications, with the intent that developers replace this with the actual communications drivers of the target platform. However, this default allows immediate testing on standard networks. In this case, the two output projects coordinate as expected. The node with `ReadThread` reads example radar images and passes them to the other node *now running in a separate JVM* on which `quantizeThread` and `dctThread` process them. `outputStage` displays the processed images. The two output binaries can be placed on separate networked computers with the same functional behaviour. The single input program has been automatically converted into a networked program according to the allocation diagram in the system model.

## *6.7 Analysis of Deployed Code*

During refactoring, AnvilJ constructs a minimal runtime to support each output project and refactors the code to use this runtime. The refactoring engine reports all changes it is making to the input code for each output project so that the generated code can be traced back to the input code. These changes are very small and only occur at well-defined points. For example, these lines appear at the start of the `run()` method of the `Main` class of the input software:

```
readThread.start();
dctThread.start();
quantizeThread.start();
```

After refactoring this becomes:

```
//Instantiate the Object Manager for node id 0
anvilj.refactored.Globals.om = new anvilj.ObjectManager(
      new anvilj.Settings(true, false, false), 0,
      new anvilj.socketcomms.SocketComms(0),
      new anvilj.refactored.Architecture(),
      new anvilj.refactored.ThreadCreator(),
      new anvilj.refactored.SharedMessages(),
      new anvilj.refactored.Routing());
anvilj.refactored.Globals.om.start();

readThread.start();
//Start thread id 1 on node id 1
anvilj.refactored.Globals.om.startThread(1, 1);
//Start thread id 2 on node id 1
anvilj.refactored.Globals.om.startThread(1, 2);
```

This code sets up and initialises the Object Manager (OM, AnvilJ's runtime support) for the current node. The implementation of the OM is automatically generated in the `anvilj.refactored` package and is unique to each processing node of the final system. For example, the AnvilJ `Routing` object contains routes to the other nodes of the system with which this OM will need to communicate. Nodes that it does not communicate with are not detailed. If the code is updated then more or fewer routes may be added, but this will always be a minimal size. Routes are planned offline according to the detailed hardware specification diagram.

Note that two of the calls to `Thread.start()` have been rewritten by the refactoring engine to calls into the OM. This is because the threads `dctThread` and `quantizeThread` are allocated to another processing node, so they are started by calling into the AnvilJ runtime. The runtime sends a 'start thread' message to the processing node that hosts the given thread. The call to start thread `readThread` has not been translated, however, because it is allocated to the current node. If the allocation diagram is altered and AnvilJ is rerun, the refactored calls will change.

## *6.8 Retargeting for New Platforms*

Retargeting the case study for a new architecture is simply a case of preparing a new detailed hardware specification diagram and amending the allocation diagram. Figure 17 shows a revised target architecture. This is the same as the original case study architecture (shown in figure 13) however a third processor has been added. The revised allocation diagram allocates the threads more evenly and can be seen in figure 18.

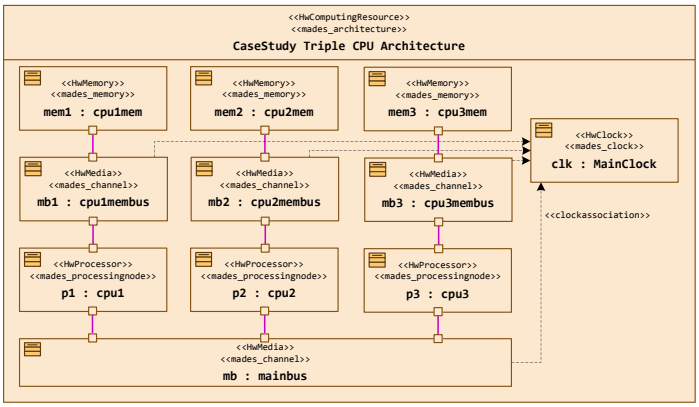


**Fig. 17** Revised hardware specification diagram for the case study target architecture.

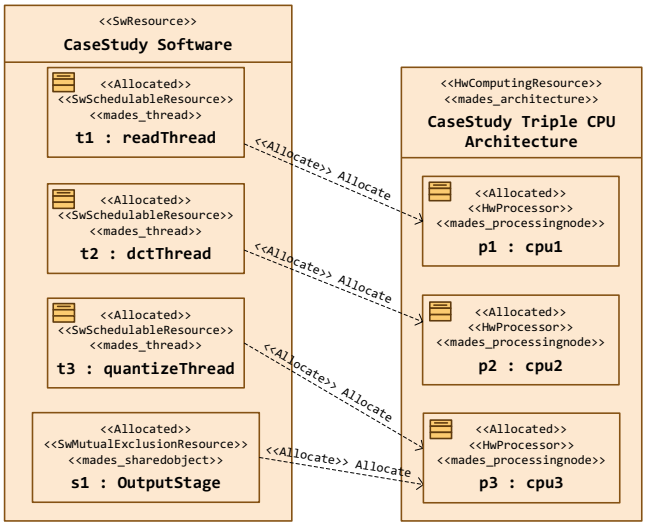


**Fig. 18** Revised allocation diagram for the case study.

Once the model has been updated, it is re-exported as XMI and AnvilJ re-run. As the hardware diagram now contains three processing nodes, this produces three output projects with the AnvilJ Instances distributed as described by the allocation diagram. Once again, initial functional verification can be performed by executing the three output projects and observing that the functional behaviour is again identical.

## 7 Conclusions

This chapter has presented some of the major problems encountered when developing complex embedded systems. The hardware architectures of such systems are characterised by the use of non-standard, application-specific features, such as multiple heterogeneous processing units, non-uniform memory architectures, complex interconnect, on-chip networks, and custom function accelerators. These features are poorly supported by the programming languages most commonly used by industry for embedded development (such as C, C++ and Java) because these languages assume a 'standard' architecture with a simple programming model. Furthermore, many embedded systems are real-time or safety-critical systems and so are subject to many additional restrictions that affect the development process. Existing approaches to solve these problems tend to lack industrial support; either because they complicate certification through the use of new languages and tools; because they prevent the use of legacy code; or because they are not integrated well enough into existing development processes.

The chapter then described AnvilJ, a novel approach for the development of embedded Java. Unlike most virtualisation systems that operate primarily at runtime, AnvilJ operates primarily at compile-time and uses a restricted programming model based on a technique called Compile-Time Virtualisation. This restricted model allows AnvilJ to operate with vastly reduced runtime support that is predictable and bounded. In addition, whilst the CTV model imposes restrictions on the programmer, these are shown to be less than is imposed by most real-time development processes.

In order to aid industrial acceptance, AnvilJ is integrated into a model-based engineering tool flow as part of the MADES project using traceable model transformations implemented in the Epsilon framework. MADES' modelling language is augmented with a small set of stereotypes to provide the additional modelling information required. The use of these transformations allows AnvilJ to be used by modellers and designers without manual intervention.

The use of model-driven engineering also allows the presented approach to automate the process of hardware development. An approach is shown which translates the hardware diagrams from the system model into VHDL, a hardware description language suitable for implementation on FPGAs. Whilst this does not expose the full flexibility of VHDL or the chosen implementation fabric, it can be used for rapid prototyping, functional verification, and design-space exploration. Due to the fact that the hardware generation transformation and the software generation transformation are described by the same metamodel, the generated software will execute correctly on the generated hardware.

To demonstrate the approach, the chapter showed a case study based on the vision subsystem of an automotive safety system. The required models

are developed and passed to AnvilJ, which refactors the input code to target two different complex architectures without any code writing.

The use of AnvilJ does not make an unpredictable system predictable, however when used in an otherwise real-time development process it will not make the system less predictable. In general, worst-case execution time (WCET) analysis for complex embedded architectures is a significant open problem. Almost all of the schedulability and WCET analysis performed for uniprocessor systems no longer applies to multiprocessor systems and many worst-case analytical models of complex embedded hardware are still too pessimistic for real-world use. These issues are being considered within the T-CREST [42] project which aims to build a time predictable NoC based multiprocessor architecture, with supporting compiler and WCET analysis.

## References


1. J. Andersson, S. Weber, E. Cecchet, C. Jensen, and V. Cahill. Kaffemik – A distributed JVM on a single address space architecture, 2001.
2. Atego. Perc Pico. http://www.atego.com/products/aonix-perc-pico/, 2011.
3. R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES '02*, pages 73–78, 2002.
4. J. Bezivin. In Search of a Basic Principle for Model-Driven Engineering. *UPGRADE - The European Journal for the Informatics Professional*, 2004.
5. A. Burns, B. Dobbing, and G. Romanski. The Ravenscar Tasking Profile for High Integrity Real-Time Programs. In *Ada-Europe '98*, pages 263–275. Springer-Verlag, 1998.
6. A. Burns and A. J. Wellings. *Real-time systems and their programming languages*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.
7. J. Charles H. Roth. *Digital systems design using VHDL*. Pws Pub. Co., 1998.
8. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3), 2006.
9. L. R. . R. F. P. D. S. Kolovos. Extensible Platform for Specification of Integrated Languages for mOdel maNagement (Epsilon). http://www.eclipse.org/gmt/epsilon, 2010.
10. EE Times. Under the Hood - Update: Apple iPhone 3G exposed, December 2008. http://www.eetimes.com/design/microwave-rf-design/4018424/Under-the-Hood–Update-Apple-iPhone-3G-exposed.
11. J. Eker, J. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, and Y. Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.
12. D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.
13. R. Gonzalez. Xtensa: a configurable and extensible processor. *Micro, IEEE*, 20, Issue 2:60–70, 2000.
14. J. Gosling and G. Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000.
15. I. Gray and N. Audsley. Exposing Non-Standard Architectures to Embedded Software Using Compile-Time Virtualisation. *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '09)*, 2009.

16. I. Gray and N. Audsley. Supporting Islands of Coherency for highly-parallel embedded architectures using Compile-Time Virtualisation. In *13th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2010.
17. I. Gray and N. C. Audsley. Developing Predictable Real-Time Embedded Systems using AnvilJ. In *Proceedings of The 18th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2012) Beijing China, April 17-19 2012*, 2012.
18. N. Guan, M. Lv, W. Yi, and G. Yu. WCET Analysis with MRU Caches: Challenging LRU for Predictability. In *Proceedings of the IEEE 18th Real-Time and Embedded Technology and Applications Symposium (RTAS'2012)*, 2012.
19. S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. In *VLSI Design, 2003. Proceedings. 16th International Conference on*, pages 461–466, Jan. 2003.
20. J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of MDE in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 471–480, New York, NY, USA, 2011. ACM.
21. ITRS. International Technology Roadmap for Semiconductors, 2007 Edition. http://www.itrs.net/, 2007.
22. F. Jouault, J. Bézivin, and M. Barbero. Towards an advanced model- driven engineering toolbox. In *Innovations in Systems and Software Engineering*, 2009.
23. R. Klefstad, M. Deshpande, C. OŠRyan, A. Corsaro, A. S. Krishna, S. Rao, and K. Raman. The Performance of ZEN: A Real Time CORBA ORB using Real Time Java. In *Proceedings of Real-time and Embedded Distributed Object Computing Workshop*. OMG, September 2002.
24. J. Kwon, A. Wellings, and S. King. Ravenscar-Java: A High Integrity Profile for Real-Time Java. In *In Joint ACM Java Grande/ISCOPE Conference*, pages 131–140. ACM Press, 2002.
25. P. Marwedel. *Embedded System Design*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
26. T. Mattson, R. V. der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core SCC processor: the programmerŠs view. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2010*, 2010.
27. Mentor Graphics. Catapult-C Synthesis. http://www.mentor.com/catapult, 2009.
28. Modeliosoft. Modelio - The Open Source Modeling Environment. http://www.modeliosoft.org/, September 2012.
29. P. Mohagheghi and V. Dehlen. Where Is the Proof? - A Review of Experiences from Applying MDE in Industry. In I. Schieferdecker and A. Hartman, editors, *Model Driven Architecture Ű Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, pages 432–443. Springer Berlin / Heidelberg, 2008.
30. Object Management Group. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. http://www.omgmarte.org/, November 2009.
31. F. Pizlo, L. Ziarek, and J. Vitek. Real Time Java on resource-constrained platforms with Fiji VM. In *Proceedings of JTRES*, JTRES '09, pages 110–119, New York, NY, USA, 2009. ACM.
32. A. L. Pope. *The CORBA reference guide: understanding the Common Object Request Broker Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.
33. J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37:99–122, 2007. 10.1007/s11241-007-9032-3.

34. M. Rivas and M. González Harbour. MaRTE OS: An Ada Kernel for Real-Time Embedded Applications. In D. Craeynest and A. Strohmeier, editors, *Reliable SoftwareTechnologies Ů Ada-Europe 2001*, volume 2043 of *Lecture Notes in Computer Science*, pages 305–316. Springer Berlin / Heidelberg, 2001.
35. L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. Polack. The Epsilon Generation Language. In *ECMDA-FA '08: Proceedings of the 4th European conference on Model Driven Architecture*, pages 1–16, Berlin, Heidelberg, 2008. Springer-Verlag.
36. D. C. Schmidt, D. L. Levine, and S. Mungee. The design of the TAO real-time object request broker. *Computer Communications*, 21(4):294 – 324, 1998. Quality of Services in Distributed Systems.
37. Terracotta Inc. *The Definitive Guide to Terracotta - Cluster the JVM for Spring, Hibernate and POJO Scalability*. Apress, 2008.
38. Texas Instruments Inc. OMAP5430 mobile applications platform. http://focus.ti.com/pdfs/wtbu/OMAP5_2011-7-13.pdf, July 2011.
39. The Eclipse Foundation. Eclipse Java development tools. http://www.eclipse.org/jdt/, 2011.
40. The MADES Consortium. The MADES Project. http://www.mades-project.org/, 2011.
41. The Motor Industry Software Reliability Association. *Guidelines for the Use of the C Language in Critical Systems*. MISRA Ltd., 2004.
42. The T-CREST Consortium. The T-CREST Project. http://www.3sei.com/t-crest/, 2012.
43. T. Weilkiens. *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
44. D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. Brown, and A. Agarwal. On-chip interconnection architecture of the tile processor. *Micro, IEEE*, 27:15–31, Sept-Oct 2007.
45. D. Wiklund and D. Liu. SoCBUS: Switched Network on Chip for Hard Real Time Embedded Systems. In *IPDPS '03*, page 78.1, 2003.
46. Xilinx Corporation. Virtex-5 FPGA Configuration User Guide. *Xilinx User Guides*, UG191, 2006.
47. Xilinx Corporation. Embedded System Tools Reference Guide - EDK 11.3.1. *Xilinx Application Notes*, UG111, 2009.
48. Xilinx Corporation. Platform Studio and the Embedded Development Kit (EDK). http://www.xilinx.com/tools/platform.htm, June 2012.