Model-based hardware generation and programming - the MADES approach

Ian Gray*, Nikos Matragkas*, Neil C. Audsley,
Leandro Soares Indrusiak, Dimitris Kolovos, Richard Paige
University of York, York, U.K.

Abstract—This paper gives an overview of the model-based hardware generation and programming approach proposed within the MADES project. MADES aims to develop a model-driven development process for safety-critical, real-time embedded systems. MADES defines a systems modelling language based on subsets of MARTE and SysML that allows iterative refinement from high-level specification down to final implementation. The MADES project specifically focusses on three unique features which differentiate it from existing model-driven development frameworks. First, model transformations in the Epsilon modelling framework are used to move between system models and provide traceability. Second, the Zot verification tool is employed to allow early and frequent verification of the system being developed. Third, Compile-Time Virtualisation is used to automatically retarget architecturally-neutral software for execution on complex embedded architectures. This paper concentrates on MADES's approach to the specification of hardware and the way in which software is refactored by Compile-Time Virtualisation.

Index Terms—Model Driven Engineering, Epsilon, MADES, Embedded Systems, Real-Time Systems.

1 Introduction

The architectures of embedded systems are becoming increasingly non-standard and application-specific. They frequently contain multiple heterogenous processing cores, non-uniform memory, complex interconnect or custom hardware elements such as DSP and SIMD cores. However, programming languages have traditionally assumed a single processor architecture with a uniform logical address space and have abstracted away from hardware implementation details. As a result, developing software for these architectures can be challenging. Equally, such systems are frequently deployed in high-integrity or safety-critical systems which require the highest levels of predictability and reliability.

The MADES Project is an EU-funded project that aims to use model-driven techniques to enable the development of the next generation of highly complex embedded systems, whilst reducing development costs and increasing reliability. In this paper we provide an overview of the MADES approach to model-driven development of embedded systems. We also give specific details of the proposed hardware development flow and the way that embedded software can be targeted at the resulting complex architectures.

*Supported by EU Framework 7 research contract FP7-248864

1.1 MADES Project Goals

The MADES project (Model-based methods and tools for Avionics and surveillance embeddeD systEmS) aims to develop the elements of a fully model-driven approach for the design, validation, simulation, and code generation of complex embedded systems to improve the current practice in the field. MADES differentiates itself from similar projects in that way that it covers all the phases of the development process: from system specification and design down to code generation, validation and deployment. Design activities exploit a dedicated language developed on top of the OMG standard MARTE (Modeling and Analysis of Real-time and Embedded systems) [10], and foster the reuse of components by annotating them with properties and constraints to aid selection and enforce overall consistency.

Validation activities comprise the verification of key properties of designed artifacts and of the transformations used throughout the development process, and also the closed-loop simulation of the entire system. Code generation addresses both conventional programming languages (e.g., C) and hardware description languages (e.g., VHDL), and uses the novel technique of Compile-Time Virtualisation to smooth the impact of the diverse elements of modern hardware architectures and cope with their increasing complexity.

All these aspects will be fully supported by prototype tools integrated in a single framework, and will be thoroughly validated on real-life case studies in the surveillance and avionic domains. The project also aims to develop a handbook to provide detailed guidelines on how to use MADES tools in the development of embedded systems and promote their adoption.

2 Overview of the MADES Approach

This section provides an overview of the model-based hardware generation and programming aspects of the

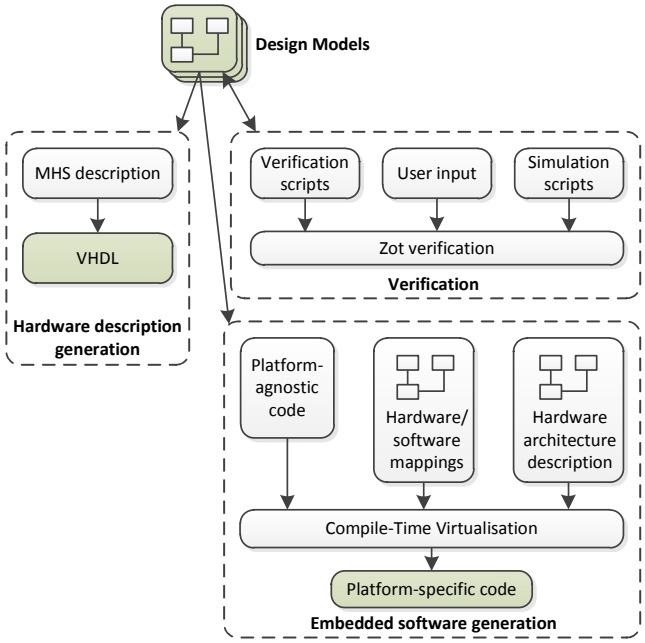

Fig. 1. Overview of the artifacts in the MADES approach

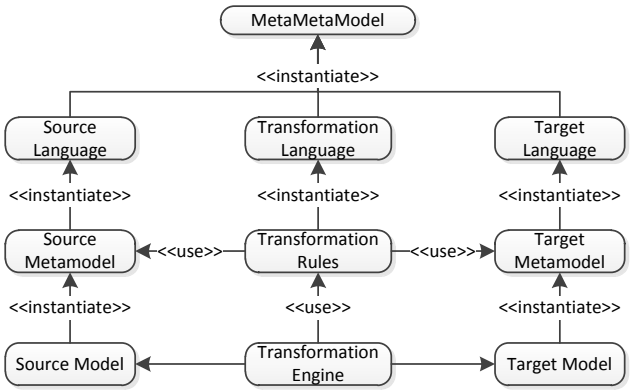

Fig. 2. MADES Mapping Scheme

MADES project and identifies the high level artifacts and the mappings between them. A conceptual model of the different inter-relationships between the various artifacts of the approach is illustrated in figure 1.

One of the main characteristics of the MADES approach to software development for embedded systems is that it is model-driven and transformation-based. Model transformations (also called mappings) are used to transform one or more input specifications into one or more output specifications. Model transformation languages are defined in a metamodel level and establish the relationship between source metamodel elements and target metamodel elements (see figure 2).

From figure 1, it can be seen that the MADES model-driven approach focusses on three areas:

- Generation of platform-specific embedded software from architecturally-neutral software specifications.
- Generation of hardware descriptions of the modelled target architecture.
- Verification of functional and non-functional properties.

Since the MADES approach is model-driven, every development effort starts with building the relevant design and analysis models. These models are expressed in the MADES modelling language and they are considered as first-class artifacts. As such, they guide the rest of the development process by being used as input to the various tools of the MADES toolset such as the verification and code generation tools. MADES diagrams are able to model different viewpoints and different levels of abstraction. They are able to describe the structure and the behaviour of the system, as well as time system constraints. Hence, all the system aspects of interest can be modelled using the MADES notation. Any additional information required by the transformation engines can be provided by attaching annotations on the system models or by user input.

The top-level MADES modelling language ('design models' in figure 1) is based on a combined subset of OMG MARTE [10] (a UML profile for modelling real-time embedded systems) and SysML [15] (a general-purpose modelling language for systems engineering applications). The language aims to overcome shortcomings of these existing languages and to provide:

- Specifications of functionality in an architecturally-neutral way.
- Descriptions of target hardware.
- Deployment diagrams that map functionality to hardware/software.
- Timing and non-functional properties for early and frequent verification.
- Code-reuse, component-based design, and maintainability.

A full specification and motivation for the MADES language is outside of the scope of this document and will be presented in a dedicated paper at a later date.

Verification and simulation play a key role in the MADES approach. Such activities are present during:

- Verification of key properties on designed artifacts (for example, whether a system will meet a specified deadline, or be able to support a specified volume of data).
- Closed-loop simulation based on detailed models of the environment (for functional testing and early validation).
- Verification of designed transformations (from high-level system models down to low-level hardware/-software implementations).

In order to provide verification and simulation in the MADES toolset, the Zot tool [11] will be used. There are three kinds of artifacts that are required by Zot in order to perform verification and simulation. The input to Zot depends on the action the tool is required to perform. In the case of verification Zot needs a verification script and user input, which will specify the properties to be verified. In the case of simulation the input to Zot is

a simulation script. Zot uses Common Lisp as internal scripting language of the tool. The purpose of the verification phase is to provide rapid and early verification of the system being developed with the aim of reducing design time and guaranteeing correctness of the final system.

The second set of artifacts is related to code generation. The code generation phase allows the designer to model the target hardware at a high-level of abstraction and use deployment diagrams to map the input code (which is provided in an architecturally-neutral form) to elements of the hardware. Three different artifacts are needed for this. Firstly, normal architecturally-neutral code with distributed operating system features such as threading, shared memory, cache coherency etc. Currently, we are considering Real Time Java [4] as the target programming language. The benefits of using Real Time Java for real time and embedded systems development are reported in the literature (e.g. [3]). Second, a target architecture description is required, which provides a high-level view of the target architecture. Finally, a set mappings which map constructs from the input software to hardware features are required. Finally, the output is a set of platform-specific programs (one for each target processor), written in the same language as the input architecturally-neutral code, i.e. Real Time Java.

Finally, the last artifact of the MADES approach considers generation of synthesisable hardware descriptions from the hardware model. The deployment mappings of the code generation phase use a high-level description of the capabilities of the desired target architecture (for example, "three processors connected with a common bus, two banks of shared memory"). This can be reified into an unambiguous hardware description for implementation using the MADES approach.

2.1 Related work

Model-Driven Engineering (MDE) is a development paradigm which has been advocated as an effective way to deal with the increasing complexity of embedded systems. MDE promotes the use of models at different levels of abstraction and transformations between them in order to drive the application implementation. This section identifies other projects that also consider the application of MDE principles to the development of embedded systems.

Gaspard2 [16] is an Integrated Development Environment (IDE) for System-on-Chip (SoC) visual co-modelling. It allows modelling, simulation and code generation of SoC applications and hardware architectures. The approach shares a common philosophical background with the approach proposed in the MADES project, although the MADES approach has two unique characteristics. First, in the MADES approach the technique of CTV is utilised in order to enhance the code generation with the ability to target non-standard hardware architectures. Moreover, in the MADES approach emphasis is given to the verification of functional and non-functional properties of the system at different stages of the development process.

Another project, which applies MDE principles for the development of embedded systems, is the Toolkit in OPen source for Critical Applications & SystEms Development (TOPCASED) project [14]. TOPCASED differs from MADES as it focuses mainly on the infrastructure of an IDE for embedded systems development and not on a particular implementation. This implementation includes metamodel and model editors and a model bus, which can be used for communication with external tools.

Finally, the MARTES (Model-based Approach to Real-Time Embedded Systems) project [2] focused on how to use the standard unified modelling (UML) and SystemC hardware description languages efficiently in combination for systematic model-based development of real-time embedded systems. The results of this project have contributed to the development of the MARTE modelling standard, which is used by MADES.

3 Model Transformations

This section specifies the transformations that are present in the MADES approach between the modelling artifacts defined in section 2.

The Epsilon platform [9] is used to implement both the model-to-model and model-to-text transformations used in the MADES approach. Epsilon (Extensible Platform of Integrated Languages for mOdel maNagement) is a platform for building consistent and inter-operable task-specific languages for model management tasks such as model transformation, code generation, model comparison, merging, refactoring and validation.

Epsilon can manipulate models in any modelling language since it is meta-model-agnostic. An important feature of Epsilon is its ability to provide traceability information produced by the various transformations, which is of paramount importance for embedded systems design due to the need to comply to particular standards such as the DO-178B Standard which requires full traceability from requirements down to the source code level.

Epsilon provides different languages for model management but three of the provided languages are of particular interest for the transformation and code generation facilities of the MADES approach. These languages are the Epsilon Object Language (EOL), the Epsilon transformation language (ETL) [8] and the Epsilon Generation Language (EGL) [12].

The rest of this section details the three main model transformation chains that are used by MADES and implemented using Epsilon. This document focusses on the hardware generation and code targeting flow of MADES - other transformations are only summarised.

Design models to Zot scripts

This is implemented as a set of model-to-text transformations from the MADES language, as well as user input,

and generates Zot Scripts. These scripts could be either Verification scripts or Simulation scripts expressed in Common Lisp. To generate the Zot scripts a combination of the behavioural diagrams of the MADES language will be needed, particularly State, Sequence and Activity diagrams. A specification of the structure of the system is also needed which is derived from the Class diagram. Moreover, timing information is required, which will be derived from the Time diagram of the MADES language. Specific details on the verification transformations are outside of the scope of this document and will be covered in later publications.

Design models to architecture-neutral code

This model-to-text transformation takes as input models defined in the MADES language and it generates skeleton architecture-neutral code. MADES will concentrate on Real-Time Java as its input and output languages. This transformation should be viewed as model-assisted programming, as it is unlikely that fully-automated code generation will be used. The MADES approach will not repeat research into automatic code generation as this has been considered by many previous projects. Existing results can be incorporated into MADES. Further details of this code generation phase are outside of the scope of this document.

The code produced from this stage is architecturally-neutral, and does not consider the complex architectures that the MADES tools might be used to generate. This is refactored into architecturally-specific code for the final system using a technique called Compile-Time Virtualisation, described in section 4.

Design models to hardware descriptions

Hardware generation from source code is difficult because in general, software languages do not contain architectural information. Consequentially, this information must be obtained either through annotation (such as in user-directed co-design of Silva et. al. [13]) or through automatic synthesis (such as in the HandelC language [1]).

In the MADES approach, generation of hardware descriptions is implemented as a model-to-text transformation which takes as input models defined in the MADES language and generates a hardware architecture expressed in an appropriate hardware description language. The architectural information does not have to be inferred from the software, it is all derived from a single system model.

The MADES toolchain allows for the generation of implementable hardware descriptions of the target architecture. A chain of three hardware models are used, H2 $\rightarrow$ H1 $\rightarrow$ H0. These models are detailed in figure 3 and described below:

- *H2:* A high-level model of the target architecture. Its constituent elements are processors, memory spaces, communication channels, and custom hardware elements and it can be directly generated from the architecture description of the code generation phase (described in section 4.3). This model does not describe any lower-level details (such as addresses or bus topologies).
- *H1:* A refinement of H2, H1 is a lower-level model which codifies the bus topology of the system and specifies I/O. Does not still define the specific types of hardware instance. For example, at this level the model will still denote 'processor' rather than 'Arm9'.
- *H0:* The lowest level model that describes hardware in terms of the MHS language. Generated by model-to-model transformation from H1 using transformation rules and in-place refinement, known as 'polishing rules'.

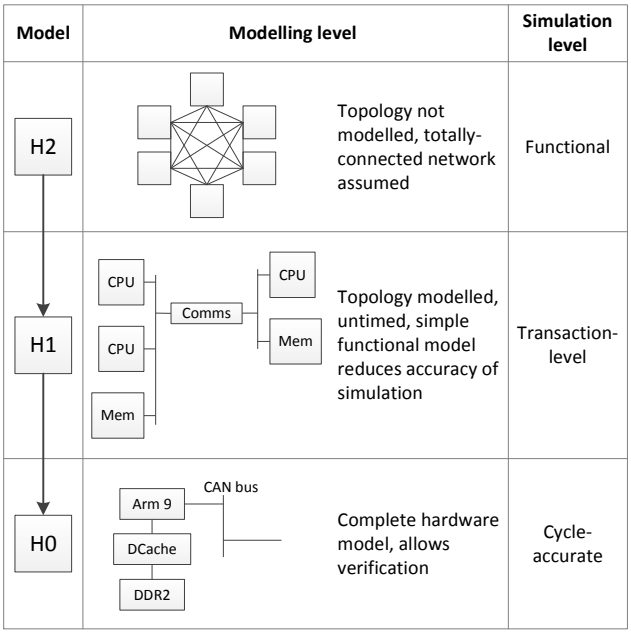

Fig. 3. Hardware models in the MADES approach

H0 demonstrates an equivalent level of abstraction to that of the Microprocessor Hardware Specification (MHS) [18] language used by Xilinx Corporation's FPGA Development Tools. MHS files are generated by the MADES tools from a H0 model using a model-to-text transformation. The Xilinx tool 'platgen' [17] is then used as an underlying HDL generator, as it can take an MHS description and generate a set of synthesisable VHDL files for implementation on an FPGA.

The use of MHS files and Xilinx FPGAs is not required by the MADES tool chain, but it is a useful language to work with as it already has robust industrial-quality tool support available. If a different language or implementation fabric is required, another model-to-text transformation can be used.

Generating a hardware description from a high-level deployment model therefore involves translating an H2 model through H1 to H0. An MHS description can then be generated from H0 using a simple model-to-

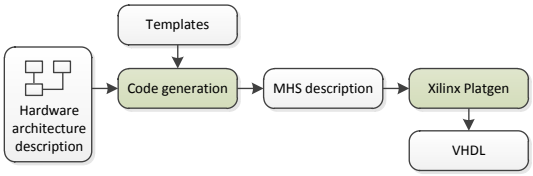

Fig. 4. Hardware transformation chain

text transformation. Moving from H2 to H0 adds extra detail, so a translation has a lot of freedom over implementation-specific details. First, hardware features (e.g. processor, memory) are translated to corresponding Xilinx IP cores (e.g. Microblaze, BlockRAM respectively). Then the connectivity of the system must be determined. H2 does not specify the target bus topology so this information is introduced by the translation to H1. This is done in two stages, memory connectivity and communications connectivity. Memory connectivity specifies how memory spaces are accessed by the processors of the system (dedicated, shared bus, etc.) and communications connectivity determines the type of interprocessor communications provided by the architecture.

Once the connected items for each processor have been determined, the processor's address map must be built. Each peripheral has a required address size, which can be determined automatically by checking the definition of the IP core. This can be performed automatically. Finally, interrupts must be connected. If a processor does not provide enough native interrupt lines then an interrupt controller instance must be created and connected appropriately by instantiating a pre-built processor-specific template. This transformation chain is illustrated in figure 4.

4 Compile-Time Virtualisation

The architecture-neutral code to architecture-specific code refactoring process uses a technique called Compile-Time Virtualisation (CTV) [5], [6]. Standard embedded software development for complex architectures requires the use of a *multi-program* development model. In the multi-program model, a complex multi-core architecture is viewed as a set of single-core architectures that interact to perform the tasks of the larger application. This is done because existing programming languages demonstrate poor support for complex, non-standard architectures. Languages tend to assume that the target architecture is a single core system with a single, contiguous memory space. Deviations from this cause the compiled code to fail, and custom hardware elements such as function accelerators are not included in the programming model so cannot be effectively exploited.

Due to this problem, the programmer is forced to provide an input program for each processor of the target system. Each program must be separately specified, developed, compiled, tested, and debugged. The problems with this model are numerous, as the programmer is forced to manually split their software requirements into a set of separate programs, which is inflexible to changing requirements, and causes many problems at system integration time. The alternative to the multi-program model is a *single program* development model, in which the operation of the entire architecture is controlled by a single program. This is the way that software developers are used to programming, supports flexible architectures and specifications, and is less error-prone.

CTV is a technique developed to support the development of software for complex embedded architectures using a single-program model without requiring the development of new languages or compilers. CTV allows the programmer to write normal architecturally-neutral code with distributed operating system features such as threading, shared memory, cache coherency etc. and to automatically distribute that program over a complex target architecture.

The mapping to the architecture can be performed *after* the program is written because the programmer's input code is architecturally-neutral. CTV allows many different code mappings to be explored quickly as it handles low-level implementation details automatically.

4.1 The CTV Virtual Platform

The layers of virtualisation and abstraction that are present in standard software development do not support changing, non-standard architectures, and they insulate the programmer from the architectural information required to efficiently software to the target hardware. CTV replaces these layers with a single virtualisation layer across the entire architecture, termed the *Virtual Platform* (VP) that contains more appropriate abstractions for embedded development.

The VP sits between a model of the programmer's input code and a model of the target hardware (or the embedded operating system if one is present). This is shown in figure 4.2. The VP's purpose is to provide the abstractions that the programmer is used to and are assumed by the programming language (such as a single shared memory space) but in a way that does not lose the architectural information required for an efficient, single-program, implementation on top of complex hardware. The VP allows the programmer to view mapping decisions at a high level of abstraction, for example where to place threads and data or which communication channels or custom hardware elements to use. The VP ensures that the low-level implementation details that result from these high-level decisions are kept hidden from the programmer.

The VP can support development of an embedded OS (so the VP sits between the hardware and the OS), or it can use a pre-existing OS as part of its implementation (so the VP sits between user code and the OS).

4.2 CTV System model

The input code confirms to the following system model, which expresses input code as three sets of objects:

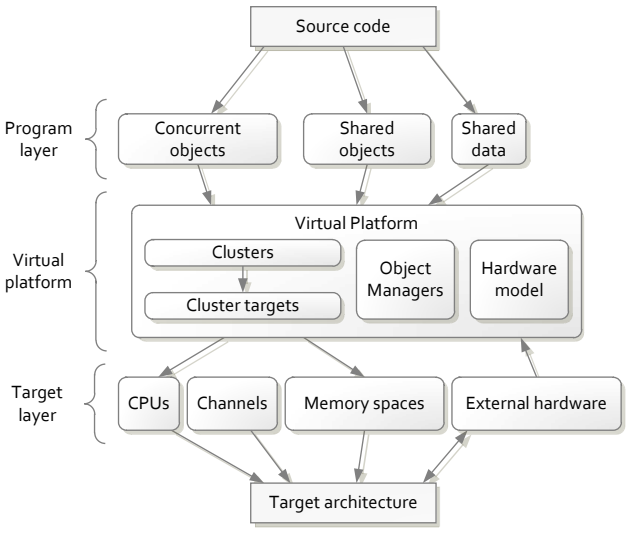

Fig. 5. CTV system model

```
ppe : processor PPE;
spe0 : processor SPE;
//and declarations for spe1, spe2 etc.

eib : channel EIB;
eib^endpoints = [ppe, spe0, spe1...];

mic : memory XDR;
mic^width = 64;
//mic^size set to the system memory capacity
ppe^memory = mic;

spe0local : memory SPE_LOCAL; //SPE local store
spe0^memory = spe0local;
//and declarations for spe1, spe2 etc.

//FlexIO access
flexio0, flexio1 : hardware FlexIO;
flexio0^ports = [ppe, spe0, spe1...];
flexio1^ports = [ppe, spe0, spe1...];

//DMA controllers
spe0dma : DMA SPE_DMA;
spe0dma^memory = [spe0local, mic];
spe0dma^ports = [spe0, ppe];
```

Fig. 6. CTV ADL describing the Cell processor architecture.

- *Concurrent objects:* A set of the units of concurrency of the source language. Concurrency is an essential modelling primitive, motivated by the trend towards highly-parallel architectures. Examples of concurrent objects are Ada tasks, Java threads and C/C++ pthreads.
- *Shared objects:* Passive constructs that implement remote procedure call (RPC) semantics that are called by the concurrent objects of the system. Shared objects allow multiple concurrent elements to synchronise with each other and coordinate their execution to avoid race conditions and the corruption of shared data. Examples are mutexes and condition variables from pthreads, protected objects from Ada, and synchronised objects in Java.
- *Shared data:* Data items that are read and written by the concurrent objects of the system.

4.3 CTV Target Architecture Description

The target architecture description provides a high-level view of the target architecture. Its purpose is to provide the CTV code generation with enough information to generation architecture-specific code from the architecturally-neutral input code described by the model in section 4.2. The target architecture is modelled using a simple Architectural Description Language (ADL) which describes four sets of elements:

- *Processing elements:* The processors of the target architecture. This set includes general-purpose processing cores, but also more application-specific cores such as DSP cores, vector processors, and SIMD units. In general, this set models the features of the hardware whose behaviour is controlled directly by the concurrent objects of the input source code.
- *Communication channels:* The set of communication channels models the data transfer mechanisms of the architecture (such as buses, on-chip networks, FIFO mailboxes, etc.).
- *Memory spaces:* The distinct memory spaces in the system. Memory spaces may contain data, code or both, according to the architecture of the processing elements used. The compiled code of concurrent objects and shared objects and the bit-level representations of shared data items are stored in the memory spaces of the system. Caches are not described as a separate memory space because their use is transparent to software and the processor.
- *Other hardware elements:* The other hardware elements set models features of the target architecture that are accessible from the processing elements of the system but are not expressed by the previous three sets. This includes I/O devices, which are required for interfacing with the outside world, and application-specific hardware such as function accelerators, radio transceivers or real-time clocks.

An example ADL to describe the IBM Cell processor architecture [7] is shown in figure 6.

4.4 CTV Outputs

From the inputs defined in the system model of section 4.2, CTV generates two sets of output code.

- *Processor-Specific programs:* Recall that CTV allows software development using a single-program model. Accordingly, the CTV code generation process must take the single input program and split it into a set of programs, one for each target processing element. This is described in section 4.4.1.

- *Architecture-Support Libraries:* The processor-specific programs generated by CTV also require the presence of a set of libraries that interface with the low-level target hardware and provide distributed language services (such as coordination, threading, memory management and migration). This is described in section 4.4.2.

4.4.1 Processor-Specific Programs

According to the provided hardware/software mappings, the input program is split into a set of programs, one for each processor of the system. Each processor-specific program is a small subset of the input code, usually the body of the thread that is mapped to the current processor and whichever library functions are called. This can be determined using reachability analysis and call-graph generation. The processor-specific programs must also be refactored so that they make use of a set of architecture-support libraries that are generated alongside and detailed in the following section.

4.4.2 Architecture Support Libraries

The processor-specific programs will not correctly execute because they are written using a standard programming model. Normal development languages (such as Java, Ada and C) assume a shared global memory space, universal communications and coherent caches. None of these assumptions are likely to be the case in a complex, non-standard, embedded architecture. To solve this problem, three support libraries are generated by the refactoring engine according to the target architecture and hardware/software mappings to make efficient use of the underlying hardware. The generated libraries provide:

- A distributed communications library which allows the transfer of messages between the processors of the system using the communication channels in the target architecture description. All other libraries use this layer as a base to implement their higher-level algorithms.
- A shared memory system which handles remote data access, cache coherency and data marshalling. The library provides a distributed solution that avoids a single point of contention.
- Distributed language features that implement system-wide features of the source programming model such as inter-thread communications, coordination (such as mutual exclusion and signalling) and thread and data migration. The specific features that are provided are determined by the source language of the input program. Simpler languages like C do not require as much run-time support as a more high-level language like Java or Ada.
- Driver code for custom hardware elements to allow the generated libraries to manipulate the target hardware.

A version of the libraries is constructed for each processor in the system. These processor-specific libraries are optimised to use the memory, communications and hardware available to the processor.

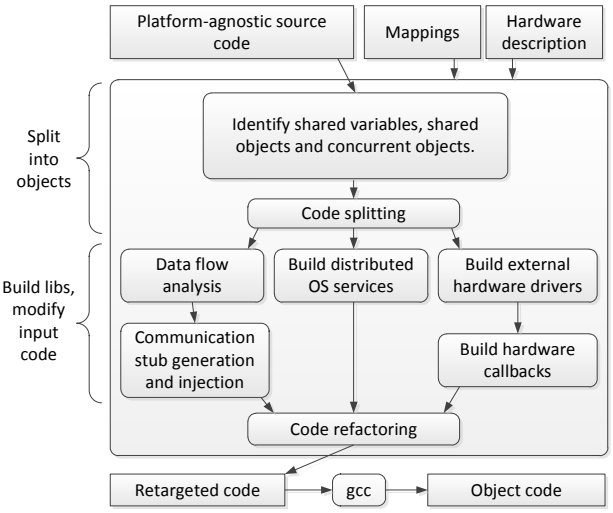

Fig. 7. The CTV refactoring process

4.5 CTV Implementation

CTV aims to produce a low overhead implementation that is suitable for use in a resource-constrained embedded system. The process can be broken down in the steps illustrated in figure 7 and detailed in the remainder of this section.

4.5.1 Parse Input Program

This stage must parse the incoming source code so that it can be represented in terms of the CTV system model (detailed in section 4.2). This involves extracting the concurrent objects (threads, tasks), shared objects (mutexes, protected objects, etc.), and shared data items that are present in the application.

4.5.2 Code Splitting

The code splitting stage is the phase of the refactoring process that splits the input program (expressed using a single-program model) into a set of processor-specific output programs, one for each target processor of the system. The hardware/software mappings provided to CTV state which processors the threads of the system should be mapped to, and to which memory spaces items of shared data should be mapped.

The callgraph of the input program is built and used to perform reachability analysis in order to determine the subset of the input program, which is accessible, by each thread. In this way, each processor-specific program will only contain the code for threads mapped to that processor, and any shared libraries that those threads access. It will not contain the code of the other threads of the system.

If multiple threads are mapped to the same processor then the use of an embedded microkernel is assumed to provide threading and scheduling services.

4.5.3 Build Architecture-Specific Libraries

The output programs of the code splitting stage will not execute correctly if they reference non-local shared memory or use language features, which the language runtime does not expect to be distributed over a complex architecture (such as threading, communication etc.). To solve these problems, four architecture support libraries are built that handle these problems. Then, the code of each processor-specific program is refactored to make appropriate use of the newly created libraries. The libraries are described in the following sections.

4.5.4 Communications Layer

The VP generates a universal communications layer that allows all threads to communicate transparently (which is assumed by all modern concurrent languages). The communications layer is generated at compile-time, optimised specifically for the current target architecture.

The layer is implemented with a compile-time multi-stage permutation routing system. Each processor of the system that is involved in a communication has a small communications kernel added to its interrupt handler to allow the processor to fetch and forward messages. In complex architectures, some processors will be used as routers, forwarding messages between otherwise separate areas of the architecture.

Routes are calculated offline between static elements so impose minimal overhead. Dynamic elements (such as migrating threads) require that they update the communications layer each time they migrate.

4.5.5 Shared Memory System

The primary goal of the shared memory system is to allow the VP to present a single logical address space to the source language because this is the memory model assumed by most standard development languages. This requires the implementation of an object-based distributed shared memory system that can allow the threads of the system to share data efficiently and coherently. In all cases it is assumed that the program already exhibits correct concurrent behaviour without race conditions. The library is required to ensure correct behaviour in the following two situations:

- A thread is accessing data that is not locally addressable (i.e. it is not directly connected to the memory bus of the processor) and may be shared between multiple threads. The system must cooperate to pass data between the processors of the system to give the illusion that all data is available to each thread.
- A thread is accessing data, which is locally addressable, but is shared between multiple processors. Caches coherency must be considered in this case.

The generated library implements algorithms to pass shared data elements using the communications layer. Small communications kernels added to the processors of the system cooperate to provide a distributed system without bottlenecks that can scale to large multicore systems.

4.5.6 Distributed OS Services

The generated code must also provide services that are presumed by the source programming model. Recall that CTV is a language-agnostic technique and therefore may be used on a range of languages. When used with a C-POSIX application, the VP is required to expose POSIX constructs that are implemented over the target architecture. In practice, this requires implementation of inter-thread communications and coordination features such as mutexes and condition variables. In a language such as Ada, the language model requires distributed implementations of the inter-task rendezvous, protected objects, and other such features. This stage of library generation is therefore dependent on the chosen target language. The communications layer and shared memory system generated previously are used to provide the majority of the implementation.

4.5.7 Hardware Drivers

The final generated library involves the inclusion of drivers for custom hardware elements that are exposed as part of the VP. Driver code is not generated by CTV; instead a library of pre-written drivers for common hardware types is consulted. Drivers for the hardware elements that are part of the target architecture are included in the generated code. The drivers are used by the communications layer to operate the communications channels and I/O devices of the target system.

4.5.8 Refactoring

The refactoring stage operates on the set of processor-specific programs (the output of the code splitting stage). Recall that these programs will not execute correctly if they reference non-local shared memory or use language features, which the language runtime does not distribute, over the target architecture. The final code generation stage must therefore refactor the processor-specific programs to call the architecture-specific libraries, which implement these features. This is done in three phases:

- *Shared memory:* The refactoring process analyses the source code of the processor-specific programs for locations where shared data is accessed. Calls to the shared memory library are injected at these points to fetch the shared data and coordinate with other users of the data object. Cache coherency is also considered for processors that use data caches.
- *OS services:* Any calls to OS services in the processor-specific programs are replaced with calls to the generated version that can operate over the target architecture.
- *Custom hardware:* Calls to custom hardware drivers are inserted at points where the processor-specific program interacts with any unique hardware features of the target architecture. For example, the programmer may indicate that their target architecture contains a function accelerator that they would like to access. The drivers for that feature will be

included in the stages above, and calls to those drivers injected accordingly.

4.5.9 Compile

The output of the CTV code generation phase is as follows:

- One set of source files per target processor of the original application. Generated by the code splitting phase and refactored to call functions from the generated libraries.
- One set of source files per target processor that implement the architecture-specific libraries. Includes the communications layer, shared memory system, OS services, and any hardware drivers that are used by the application.
- A custom linker script per processor generated by the shared memory system to place shared data items at the correct section of memory.

CTV does not require the use of any custom compilers or tools, apart from the CTV tool itself. The generated code is entirely compliant with the source language, so it can be compiled using that language's normal compiler (for example, `gcc` for C). Because the target architecture may contain many different types of processor, an appropriate compiler must be available that can output object code for each processor type.

The refactored processor-specific programs and each processor-specific library are all separately compiled to object files using an appropriate compiler. Then, for each target processor its object code is linked against its specific libraries using any custom link scripts generated by the shared memory library. The result of this is a single executable for each target processor. The executables can then be used in a variety of ways depending on the target application. They might be programmed directly into on-chip flash-based storage, stored on disk, transmitted into dynamic RAM during a bootstrap phase, or in the case of FPGAs, merged into the FPGA's configuration bitstream.

4.6 Example

Due to space constraints a fully-worked example of CTV is outside of the scope of this paper, but the reader is directed to existing documentation [5], [6] for code examples that demonstrate CTV's refactoring in action.

5 Conclusion

This paper has provided an overview of the model-based hardware generation and programming aspects of the MADES project. MADES defines a systems modelling language based on MARTE and SysML that allows the developer to express their system at a high-level of abstraction, and then to iteratively refine their design to reach the final implementation. MADES differentiates itself from similar work through three unique features. First, extensive use of model transformations is used to facilitate development and provide traceability. Second, verification and validation are key parts of the MADES design flow, allowing early and frequent verification of the system being developed. Third, Compile-Time Virtualisation (CTV) is used to assist the development of embedded software. CTV allows for architecturally-neutral code to be automatically retargeted for complex hardware. This paper has provided specific details on MADES's approach to the specification of hardware and the way in which that embedded software is refactored for that hardware by CTV.

References

[1] M. Bowen. *Handel-C Language Reference Manual, 2.1 edition*. Embedded Solutions Limited, 1998.

[2] M. Consortium. MARTES project. http://www.martes-itea.org/public/, 2007.

[3] A. Corsaro and D. Schmidt. The design and performance of the jrate real-time java implementation. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE*, volume 2519 of *Lecture Notes in Computer Science*, pages 900–921. Springer Berlin / Heidelberg, 2010.

[4] J. Gosling and G. Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[5] 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.

[6] 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.

[7] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[8] D. S. Kolovos, R. F. Paige, and F. A. Polack. The epsilon transformation language. In *ICMT '08: Proceedings of the 1st international conference on Theory and Practice of Model Transformations*, pages 46–60, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. Eclipse development tools for epsilon. In *In Eclipse Summit Europe, Eclipse Modeling Symposium*, 2006.

[10] Object Management Group. UML profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. http://www.omgmarte.org/, November 2009.

[11] M. Pradella, A. Morzenti, and P. San Pietro. The symmetry of the past and of the future: bi-infinite time in the verification of temporal properties. In *ESEC-FSE '07*, pages 312–320, New York, NY, USA, 2007. ACM.

[12] 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.

[13] E. T. Silva Jr., D. Andrews, C. E. Pereira, and F. R. Wagner. An infrastructure for hardware-software co-design of embedded Real-Time Java applications. *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 273–280, 2008.

[14] TOPCASED. The Open Source Toolkit for Critical Systems. http://www.topcased.org/, 2010.

[15] T. Weilkiens. *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[16] WEST Team LIFL, Lille, France. Graphical array specification for parallel and distributed computing (GASPARD-2). http://www.lifl.fr/west/gaspard/, 2005.

[17] Xilinx Corporation. Embedded system tools reference guide - edk 11.3.1. *Xilinx Application Notes*, UG111, 2009.

[18] Xilinx Corporation. UG642: Platform specification format reference manual. http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/psf_rm.pdf, September 2009.