Abstract

Creation of Software-Defined Radio (SDR) solutions is fraught with challenges. In the design phase, communication algorithms must be proven and prototyped. In the next phase, implementations are validated on specific hardware. In the deployment phase, verified radios must reliably inter-operate with other software. We address the challenges by threading the formal Data-Flow (DF) meta-model throughout this process, supported by high-level languages, tools and diversity mechanisms. The meta-model is stratified: kernel algorithms are expressed in plain C and separated from dependency graphs expressed in a Domain-Specific Language (DSL). The separation is transparent: kernels are used “as-is” in the final application. We apply generative techniques to recover elided communication & synchronization code from abstract models and illustrate the benefits of our approach for streaming applications such as the Digital Audio Broadcast (DAB).

Categories and Subject Descriptors C.3 [Special-purpose and Application-based systems]: Real-time, embedded and Signal processing systems; D.1.2 [Automatic Programming]: Automatic analysis of algorithms, Program synthesis, Program verification; D.1.3 [Concurrent Programming]: Parallel programming; D.2.6 [Programming Environments]: Integrated environments; D.3.2 [Language Classifications]: Concurrent, parallel and Data-flow languages; D.3.3 [Language Constructs and Features]: Abstract data types, Control structures, Concurrent programming structures; D.3.4 [Processors]: Code generation, Compilers

General Terms Algorithms, Design, Languages

Keywords Embedded Software Engineering, Software-Defined Radio, Language Engineering, Generative Programming, Diversity Management, Model-driven Software Development

1. Introduction

At Next Experience Semiconductors (NXP), we are looking into the implementation of modern broadcasting and communication standards in the automotive domain: DAB, Digital Video Broadcast (DVB) and Wireless LAN (WLAN). All of these radio standards exhibit streaming DF properties. This entails application decomposition into a set of independent actors (nodes), connected together by a set of channels (edges), thereby forming a DF graph. The use of a particular Model of Computation (MoC) - the Synchronous DF (SDF) [20], which enforces a specific synchronization & communication discipline on each actor via firing rules [18] is widespread in embedded real-time domain, which encompasses digital communication systems used in the automotive domain.

In addition, these radios must coexist (though not all simultaneously) on multi-core radio hardware platforms of NXP. These contain a mixture of ARM, Digital Signal Processor (DSP), and custom hardware such as tuners and accelerator cores. We have legacy Operating System (OS) run-time infrastructure that is on one hand tuned to this multi-core hardware and on the other hand supports multiple software implementations of DF applications efficiently.

Traditionally, the software construction process in the business follows one of the modified waterfall [24] development models. With SDR, it is aggravated by multiple stages in the implementation, where radio algorithms are: (1) specified mathematically in the standard, (2) prototyped using MATLAB, (3) validated in fixed-and/or floating-point C code and then (4) translated either into optimal implementations in an Application Specific IC (ASIC) or Field Programmable Gate Array (FPGA) hardware, or in software using DSP optimizations. Every step of this process must be designed, implemented, verified, and maintained. This presents a sizeable problem to the industry, which must avoid the exploding costs of product deployment in the face of many emerging new standards.

The inherent computational complexity of dealing with multiple modems (implementing the inner receiver functionality: tuner, demodulation, equalization, tracking), codecs (implementing the outer receiver functionality: deinterleaving, de puncturing, decoding, descrambling) and the computational complexity of dealing with noisy communication channels, necessitates the use of multiple cores and hardware accelerators in one platform. Besides this, SDR solutions must also provide soft- and hard real-time guarantees, sometimes even when running in parallel with other (control) tasks. To cope with these challenges, we introduce LIME - a Parallel Programming Model (PPM) specifically targeted at multi-core embedded real-time SDR architectures, in Section 2. We discuss its language-level interfaces, diversity management, code- and document-generation in Section 3. Finally, we offer conclusions and sketch future work in Section 4.

1.1 Related work

Providing correct-by-construction software in general, and SDR in particular, has been the goal of much research [12] [6] [14] [28]. Various prospective solutions have been proposed (we mention some, but make no attempts at categorization):

Model-Driven Software Development (MDSD) promotes top-down partial development of applications from formal, often visual models, that are used to automatically derive code skeletons, which are subsequently merged with other, non-modeled, custom source code by hand.
Aspect Oriented Programming (AOP) \cite{17} offers a way to enforce correctness by applying \textit{advices} to certain points in the source-code, as defined by \textit{point-cut} rules.

Generative Programming\cite{19} approaches this problem from the bottom-up. A \textit{higher-level} variant, or possibly, another programming language entirely is used for code-generation. If the target language is already sufficiently high-level (e.g., higher-order), this is known as \textit{meta-programming} by \textit{multi-staging} & \textit{off-shorin}.

Generic Programming advocates the use of reflective methods to obtain the most general implementations, which still must be automatically specialized by tools to recover efficiency.

\textbf{TEMPLATES}\cite{10} offer a type-directed way of generating code within a single language by a limited form of meta-programming based on template parameter substitution (essentially, \textit{β}-reduction).

Unfortunately, this goal has still been elusive, in the general case. AOP, for example, is constrained by obfuscation of the original models by concrete source-code. Templates are often not expressive enough for the problem domain. And generic programming delegates the problem to tool vendors, which must either recover domain-specific optimizations from very high-level programs (e.g., via \textit{partial evaluation} or \textit{super-compilation}) or rely on very complex compile-time annotations such as type-classes in Haskell.

A common path proposed in an academic setting \cite{26} and validated by the software engineering community \cite{6,12} is embodied by the application of modeling techniques of visualization and transformation. This typically encompasses usage of MDSD methodologies such as: Universal Modeling Language (UML), Model-Integrated Computing (MIC) \cite{13}. Eclipse Modeling Framework (EMF), Atlas Transformation Language (ATL).

Although some standardization has taken off in the Object Management Group (OMG) in form of Query/View/Transformation (QVT) languages, they have not been applied to DF yet. On the other hand, most work on new DF languages \cite{1} and MoCs \cite{2} is done in an academic setting, rather than in an industrial setting: integration of MDSD with formal meta-models and code generation for cost-efficient embedded real-time platforms has not yet been taken further than some early work on MIC. Although existing proprietary solutions such as MATLAB Simulink \cite{4} could be used for this purpose, the mismatch in scope (dynamic vs. static, control-flow vs. data-flow) would result in needlessly complex designs that are also difficult to comprehend.

Finally, the way these approaches apply to concrete applications is highly dependent on the domain. In this paper, we consider the domain of SDR, an emerging field in which aspects of (1) embedded \textit{real-time} systems, (2) parallel \textit{multi-core} programming and (3) Component-Based SW Engineering (CBSE) engineering are combined.

\subsection{1.2 Problem statement}

C and Java are the two most commonly used languages in embedded software industry in general and in automotive in particular. Often, these languages are used for implementing applications with use-cases having conflicting requirements and/or making complex trade-offs (see subsection below). Because both of these language families are statically compiled and because both are deeply rooted in the imperative domain, they lack sufficient abstraction capabilities to accommodate such complexity by themselves, without resorting to external generative methods. Although object-oriented methods in general, and the Java language in particular, does offer a better way for \textit{encapsulation} via subtype polymorphism, it only addresses some problems well, leaving concurrency and communication issues largely to the hands of the implementors. Also the C++, which does support powerful template mechanism, is yet to see widespread use in the embedded industry. This is mainly due to its syntactic and semantic complexity.

That is why a meta-tool, i.e., a pre-processor, is frequently needed to translate (“lower”) a more expressive source language to the target imperative language. This reminds us of the process where a high-level imperative language is being translated to a machine language by a compiler. Obviously, one can’t expect an average software engineer to become a specialist in compiler technology, which nevertheless often becomes necessary when one starts building a DSL with e.g., the C pre-processor. The pitfalls of using it at all are extensively studied in \cite{11}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1.png}
\caption{Traditional vs. proposed approach.}
\end{figure}

Practical usage of existing MDSD techniques exposes an impedance mismatch between model-based development and the level of code-generation required for cost-efficient automotive systems. Although the automation of skeleton code generation from existing meta-models is initially advantageous, the need to semi-automatically entangle customization into generated code quickly becomes a burden. Last but not least, the documentation has to be brought “up-to-date” with the code & the model as well, complicating the task of providing a traceable \textit{end-to-end} solution. This situation is depicted on the left of Fig. 1.

The application of formal modeling also requires the automotive software industry to radically change their way of working. Most developers are currently trained in either hardware design, communication algorithms and MATLAB, or in embedded programming with C and have little, or no experience in formal models or methods for software construction. In addition, the tension between tool vendors and respective standardization, as always, impedes seamless interoperability in practice.

\subsection{1.3 Our research}

LIME addresses the following software requirements on SDR architectures that we focus on in this paper. Most of these are corollary to CBSE requirements \cite{27} applied in SDR context \cite{29}:

1. support for \textit{stream-based} processing, i.e., repeated computation using (practically) infinite data-streams
2. provide \textit{efficient} mappings of DF graphs on resource-constrained platforms found in the automotive domain
3. the programming style must be \textit{intentional} \cite{25}, i.e., describe signal-processing functions directly in terms of the Data-flow
4. the input must be both human- and machine-readable. This implies a \textit{terse} and \textit{expressive} textual format, e.g., for automated merge conflict resolution with a Version Control System (VCS)
5. the kernel algorithms must be specified independently from the underlying \textit{interconnect} and at a high abstraction level. Following best-practices in the industry, one should not deviate too much from C, preferably, as close to C as possible
6. it must be possible to facilitate reuse of multiple radio (instances) across multiple platforms by:
   a) allowing creation of multiple instances of an \textit{actor} (making life-cycle management of \textit{state} explicit),
   b) specification of start-up as well as shut-down behavior of application graphs (defining \textit{explicit} constructors & destructors),
7. model instances (as well as executables and documentation) must be derivable directly from the input. To reformulate this requirement in the terms of traditional DF analysis techniques: (i) specify DF delays as initial tokens, which is needed for static analysis of transients (see Subsection 2.2), and (ii) express restricted forms of control-flow in addition to SDF, e.g., Cyclo-Static DF (CSDF) [3] and Boolean DF (BDF) [9].

It is of outermost importance to obtain a language for expressing actors, channels and graphs that integrates well with the rest of the system: Design/Documentation systems, Software Development Environment (SDE), Integrated Development Environment (IDE), VCS, OS run-time infrastructure, underlying C language dialect and specific compilers. C is still the de-facto standard in embedded industry, and of course we are not in a position to develop a complete solution from scratch - we have to deal with many existing tools and entrenched methodologies.

2. Less Is More (LIME) PPM

At the heart of our approach lies the idea to segregate the hardware architecture and application-specific communication & synchronization details (i.e., dependencies) from computational kernels. LIME PPM [15] implements this idea and proposes to keep C as the language of choice for (essentially) sequential kernels, while at the same time abstracting away the necessary plumbing for internal and external connectivity of the algorithms behind a simple, declarative façade. It comes in several flavours: as a graph language expressed using an Extensible Markup Language (XML) schema and as a visual, diagrammatic language (see Fig. 2). The concept of a port defined at the boundary between the computation and communication/synchronization strata is the key for powerful expression of data-, output- and anti-dependencies. Tangentially, control dependencies are also addressed, but via actor roles.

![Figure 2. Sketch of a simple graph.](image)

The signature of an actor provides information on the (1) role/behavior of the actor, (2) data-type, (3) data-rate, (4) direction of access and (5) synchronicity of each port as well as additional attributes for expressing packetized vs. streaming Input-Output (IO), speculative vs. irrefutable execution and deterministic vs. non-deterministic choice as well as a number of other communication patterns and skeletons.

Similarly to Haskell, which separates purely functional code (side-effect-free computations) from effectful code (e.g., monadic IO), we untangle code into computation and dependency strata. Unlike Haskell, however, the former is expressed directly using a restricted subset of C while the latter is reflected as IO dependency patterns in the graph specification. The patterns are eventually refined as system-specific communication & synchronization primitives which are later tangled together transparently with (largely unmodified) computations by a specific Back-End (BE) code generator. Each platform supported by LIME provides a specific implementation of such a slimer code generator, each implementing a standardized Command Line Interface (CLI).

A tool that implements such model→source transformation has the sufficient knowledge to automatically group and map actors to OS-specific tasks, threads, interrupts and start- and clean-up code (i.e., schedule them), optimize allocation of buffer and state memory as well as introduce OS-specific plumbing and instantiate a number of other common skeletons in parallel programming such as broadcast/reduce and scatter/gather.

2.1 An example

LIME can be used to expose most concepts from the streaming DF domain concisely. Below is a model that contains a source actor with an out-port, (b) copy actor with in-port and an out-port and a (c) sink actor with an in-port. These actors are connected using edges typed “fifo” in Fig. 2. Although the exact implementation of communication channels is of course hardware-dependent and subject to specific optimizations, this does specify the expected First-In First-Out (FIFO) behavior of ports connected by such edges.

![Figure 3. source, copy and sink actors.](image)

Complete sources for these modules are using the K&R syntax in Fig. 3 which largely makes stylistic sense for this application. The DF graph sketched in Fig. 2 together with these modules, provides enough information for our compiler called slimer to generate (1) the actual platform-specific shells, (2) the OS configuration, and (3) startup code to run the streaming graph on an embedded parallel platform. The following syntactic properties of LIME can be directly observed from this simple application, which (for clarity) uses some syntactic sugar that is summarized in Table 1.

- **communication & synchronization** is implicit: edges of the graph denote, but do not specify the exact operations needed
- data-dependencies, **ports**, are out-ports by default while in-ports are denoted by the `const` qualifier
- degree of dynamism and exact data-rates are explicit as array type qualifiers and argument modifiers
- the `restrict` keyword indicates that the port is streaming; its contained data may not be aliased as is the case for “state” ports

![Table 1. LIME C99 syntactic sugar.](image)

2.2 DF Analysis

The SDF analysis can be performed directly in terms of the DF meta-model [19]. A model instance of some algorithm is reflected as data-dependency edges (channels) E between functional units (actor nodes) N. Typically, each node nᵢ ∈ N specifies a kernel function K(nᵢ) = {fᵢ}, although in general there may be several functions encapsulated in one unit, as in CSDL. Also, a role, which with every port participates in a channel eᵢ ∈ E is declared for each node nᵢ. A port can be either a data **provider** i.e., out-port, or a data **consumer** i.e., in-port. In addition, the data rate for each port is statically determined: Rate(nᵢ, eᵢ) = const. It is apparent
that this formulation is captures all of the, and only the information from the Fig. 2 (edges) and Fig. 3 (implementation in C). Analysis of this stratified model begins with construction of an incidence matrix:

\[ e \in E, n \in N, \Gamma_{e,n} = \begin{cases} \text{Rate}(n, e), & \text{bytes produced; or} \\ -\text{Rate}(n, e), & \text{bytes consumed} \end{cases} \]

One of the most important properties of DF models is liveliness (the absence of deadlocks) and boundedness (ability to run the model indefinitely in finite space). A deadlock in DF implies that no actor function can be activated because there is no input data available, or because no space for output data is available. Boundedness implies that the buffer-space that is allocated at compile-time is never exceeded at run-time.

To ascertain these properties, each channel is assigned some memory, with an upper-bound on its size: \( \forall e \in E : b_e \geq 0 \). In addition, for graphs that contain cycles, some edges must contain initial tokens, i.e., be delayed before steaming can start: \( \exists e \in E : b_e > 0 \) (e.g., this is most apparent for “state” edges which must be initialized before steaming starts).

Given this information, we can calculate the current channel occupancy for each new \((i + 1)\) activation of an actor \( a \) as follows:

\[ b_{a,i+1} = b_{a,i} + \Gamma_{a} = \Gamma_{a}^{i}, \]  

where \( a \) is a unit vector for actor \( a \).

The objective of the analysis is thus to find \( r = \sum_{\lambda} r_{e} \) such that: \( \Gamma r = 0 \), satisfying the boundedness requirement (given any \( b_{i} r_{i} \), activations of each \( a_{n} \) in period \( r \) do not overflow the memory). In addition, a fix-point \((b_{i})\) of the \( \Gamma_{i}\) (recurrence above applied \( |\pi| = \text{const} \) times) must be found. Then, the transient sequence of \( b_{0} . . . b_{\infty} \) must be verified for feasibility, which means having no negative buffer occupancies. This analysis can be performed using standard linear-programming techniques, and the sequence \( 0 . . . s \) (which can be arbitrarily long, if the model contains cycles) can be determined by simulation.

<table>
<thead>
<tr>
<th>Type NAME [RAISE]</th>
<th>Containing Type NAME [RAISE]</th>
<th>source at NODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>const TYPE NAME [RAISE]</td>
<td>N(A) points to accessible memory</td>
<td>input port</td>
</tr>
<tr>
<td>volatile TYPE NAME [RAISE]</td>
<td>N(A) points to current accessible memory</td>
<td>input port</td>
</tr>
<tr>
<td>TYPE NAME [RAISE]</td>
<td>N(A) points to volatile accessible memory</td>
<td>input port</td>
</tr>
<tr>
<td>TYPE NAME [RAISE]</td>
<td>N(A) points to non-accessible memory</td>
<td>input port</td>
</tr>
<tr>
<td>TYPE NAME [RAISE]</td>
<td>N(A) pointer is constant</td>
<td>shared (system) port</td>
</tr>
<tr>
<td>TYPE NAME [RAISE]</td>
<td>N(A) pointer is volatile</td>
<td>shared (system) port</td>
</tr>
<tr>
<td>TYPE NAME [RAISE]</td>
<td>N(A) may be NULL</td>
<td>shared (system) port</td>
</tr>
<tr>
<td>TYPE NAME [RAISE]</td>
<td>N(A) may be M(D)</td>
<td>shared (system) port</td>
</tr>
<tr>
<td>TYPE NAME [RAISE]</td>
<td>N(A) may be NULL</td>
<td>shared (system) port</td>
</tr>
<tr>
<td>TYPE NAME [RAISE]</td>
<td>N(A) may be M(D)</td>
<td>shared (system) port</td>
</tr>
</tbody>
</table>

2.3 Contribution

The language we propose reifies these analytical concepts, in addition to some other, more practical ones. It does so directly and transparently using C primitives (see Table 2). The standard notion of DF is restricted to “streaming” ports, as per Static Single Assignment (SSA) discipline. For efficiency reasons, we also allow “state” in- and out-ports that indicate that access to the memory involved is a restricted subset of the FIFO semantic where a single element (the state) is shared (aliased) between all activations of an actor, enforcing a sequential order of activation. With speculation, other orderings are possible. Also, we distinguish “static” (synchronous/streaming) and “dynamic” (asynchronous/sampled) ports, with the former participating in the firing rule of an actor, while the latter left under full control of the actor function itself.

Because we directly use algorithmic kernel code as one source for deriving our models (internal actor structures, roles, ports, rates etc.), with the other source being the DF graph, and because we link these kernels directly in the final binary, manual entanglement of customization code with generated code is not needed. Neither is the use of protected regions mandatory. This is what we call transparent modeling. Application of this novel technique is shown on the right-hand side of Fig. 4 (see the dotted line). In addition, we can deliver the final models, i.e., Platform Specific Model (PSM) instances together with compiled kernel code to 3rd parties and let them generate the shells, specifically optimized for their platforms.

This combination of applying transparent modeling to the kernel code (on the right of Fig. 1) and the DF analysis to communication & synchronization graphs represented by Platform Independent Model (PIM) models (on the right of Fig. 1) allows us to approach correct-by-construction SDR, where real-time requirements are guaranteed with no undue manual effort needed to port radios to specific platforms, or derive formal models for analysis.

3. Motivating example

We approach the problem of software construction by amalgamation of the novel transparent MDSD and Generate programming. In Section 2, we have already shown a formal meta-model for SDR applications supporting a number of high-level languages. These include: (1) a lower-level, machine-readable schema based on XML (2), a machine-language approaching natural languages and (3), a graphic-oriented language based on Graphviz and Hyper-Text Modeling Language (HTML) that makes a compromise between expressivity and readability. The goal in creation and evaluation of these languages is to alleviate the shortcomings of introducing a two-level, stratified DF language per se.

A visualization of one example of our machine-readable schema called the Graph Exchange Format (GXF) is given in Fig. 4. Corresponding textual representation using XML is shown in Fig. 5. These two representations are equivalent: the picture can be converted to the XML and vice versa. The size, and complexity of the textual GXF representations of real-life SDR applications illustrates the reason why we address the problem from the language-engineering perspective.

The top-level <gxf> element encapsulates a more complete example model than what was sketched in Section 2. This model instance contains a set of prototypical <node> elements, representing actor types (classes). Prototype nodes denote interfaces consisting of a number of <port> elements and specify internal structure of each node. This is because the model can be hierarchical; each top-level <node> representing a DF actor prototype can specify internal <node> elements representing contained C kernels or other contained nodes, together with their schedule.

The meta-model supports multiple instantiation: the top of the south-east quadrant contains 3 instance <node> elements, which refer to their respective classes via the stereotype attribute. Although the meta-model is tuned to accommodate singleton nodes efficiently, the need to support multiple radio (instances) on resource-constrained platforms necessitates explicit control over instantiation. These instance nodes are subsequently connected using <edge>, <from-node> and <to-node> elements contained inside of a <stream>, which serves as a scope for dependencies.

A number of auxiliary elements are shown as well, supporting traversals of the model using XPath/XLink. Most conceptual elements of the model have <id> and <type> children (also present as XML attributes). In addition, the <port> element contains (for multi-dimensional ports: a list of) <size> sub-elements and a number of other children specifying its properties (underlined elements such as const in the Fig. 5).
3.1 Towards an ideal DF language

Even with other simplifications of the markup such as [3], usage of XML for SDR in practice does not precisely capture the intent; there is inevitably some duplication of information, verbosity issues, and presence of aspects interweaved into the GFX that have less to do with the DF (e.g., XLink support). Ultimately, it is preferred that models are created, edited, and visualized in a more direct way. What way that is can be varying depending on personal experience or psychological preferences of the designer. Most commonly, this would encompass visuals (i.e., Fig. 2 and Fig. 3), but the usage of audible interfaces is also not unthinkable.

The compatibility to textual representations, however, must be maintained for traceability and integration with a VCS. Therefore, a set of bidirectional transformation tools are proposed, each addressing a specific representation variant yet compatible to the common GFX schema. In the following subsection we shall gradually introduce the various language-level engineering approaches we have tried in practice.

3.1.1 Plain English grammar

This grammar inspired by natural languages was developed for a DSL targeted at our domain. A usable variant was implemented via an ad-hoc en2xml.awk transformation tool. This employs a simple stack-based shift-reduce parser [2] that can handle inputs like the one depicted in Fig. 6.

Figure 6. Stream model.

Here, the language is structured as a set of sentences in plain English, delimited by period characters. Sentences consist of words, which are possible delimited by “(” and “)”, one of “ and “ or pairs of ‘ and ‘. The layout is free form, with underlined keywords being part of the DSL and all other entities representing themselves. Entities may be referred to another XML file, which is "opened" by take and "closed" by give, providing lexical scoping of referring regions.

The example in Fig. 6 specifies exactly the edges from the graph shown in Fig. 2, which itself is a simplification of Fig. 4 with internal {actor and FIFO structures elided. In this DSL, the actor internals such as ports, rates, roles etc. are derived from an external XML file test.gxf, which is similar to (abridged) Fig. 5 with the following subsection we shall gradually introduce the various language-level engineering approaches we have tried in practice.

3.1.2 Concise Graphviz grammar

The next language-level approach we have applied is based on Graphviz. Although the approach described in previous section is viable, the maintenance costs of transformation tools, ad-hoc parsing and ad-hoc XML generation techniques have motivated us in the direction of reusing an already existing and widely used language together with its grammar - the popular Dot scripting language used by the Graphviz project [4].

Fig. 7 shows an example of a concise Graphviz model specification that generates a GFX model like the model in Fig. 2. Although the visual, structural and documentation aspects (shape and URL attributes) do appear to be interwoven with DF aspects (record label specification and connections), this works out quite well in practice, since DF nodes, ports and arrows emerge naturally from the visual representation.

The reuse of the grammar is facilitated by re-use of the corresponding parser, which is written in C and is highly optimized to handle large graphs. At the same time, by moving to a C-centric environment, we also get interfaces to many interesting libraries such as GNU Linear Programming Kit (GLPK) for free. We use this library to solve the analysis problems from Subsection 2.2. We avoid development costs of creating portable software using C by moving to higher-level, Functional Programming (FP) environment with Bigloo: an (almost compliant) implementation of Revised® Report

Figure 7. Graphviz model.
on Algorithmic Language Scheme (RSRS). Bigloo has static typing, good Foreign Function Interface (FFI) options and supports high-level parsing with Scheme SAX (SSAX), processing with the Scheme XML (SXML) and many other useful libraries.

3.1.3 Verbose Graphviz grammar

The last approach we have tried is a variation on the previous one. Here, we try to address the limitations of the visual representation of a nodes by record shapes; something that is limited to a rigid box with sections, as depicted in Fig. 4.

![Figure 8. Visual XML specification of a model.](image)

Because Graphviz supports HTML-like labels that generalize record shapes and allow more flexible layout, it is now possible to include additional model elements quite naturally. In Fig. 8 we show a rendering of such a specification, where the internal kernel code for each node from Fig. 3 is in-place with the DF graph from Fig. 2 and where integration with HTML provides both completeness from the modeling perspective as well as better appearance.

3.2 Diversity management

Although most concerns when building correct-by-construction DF applications in the SDR domain can be addressed by language-level approach discussed above, some aspects do not fit naturally. One reason for this is that the language should ideally be platform independent, while practical resource-constrained applications can not avoid being very much dependent of platform details. Another is that we want to avoid complicating our compiler too much by trying to abstract very specific issues. It is much better to provide an orthogonal mechanism for solving these problems: diversity management. We can classify several distinct types of diversity in our domain, each of which we specifically address in our framework.

3.2.1 Platform diversity

This kind of diversity corresponds to variations in the global compilation environment. These are accounted for via declaration of compilation domain feature-tags for each component. Each tag is a C identifier, declared at the beginning of the file using `#include` TAG syntax. Because all LIME diversity mechanisms are piggy-backing on the C pre-processor, this and only this use of `#include` must be allowed. Incidentally, this is also used as a hook for the other diversity mechanisms described below. For example, on the left-hand side of Fig. 9 we can observe a component importing three features: LIME, Stdio and Assert. The first one is actually a compilation domain, therefore, we can include many dependant features by default. The last two (optional) features are specifically designed for denoted functionality (printing and debugging, respectively).

![Figure 9. Specifying domains & features.](image)

Every LIME platform may support several compilation domains (e.g., for each core present in the system), which may be used by a component via explicit feature declarations. This lets each platform’s slimer tool to supply its own sets of definitions needed to make features portable. Conversely, each component may “import” a number of specific features from a compilation domain, all of which are required simultaneously, for example, as in Fig. 4.

The feature sets may overlap, allowing portable code to be written. Essentially, this mechanism supports sharing of feature interfaces across compilation domains. For example, on the right-hand side of Fig. 4, we can observe the Stdio feature shared between the LIME and SDR domains. Having such abstract specifications allows the tool provider to deliver all the necessary tweaking as a system-supplied black box with a pre-defined interface, improving encapsulation. This simplifies the task of the implementor because he no longer has to keep track of the include files that are needed for specific features. Also, inspection of missing features can easily be done on the source-code level. A compiler error is generated if incompatible features are requested for a compilation domain.

3.2.2 Configuration diversity

The most commonly found diversity kind in embedded software is configuration. A supplier (typically: 3rd party) delivers a configurable software library, while the other party (typically: system integrator) configures the library and integrates with other component configurations. More often than not, this entails (1) changing supplied source or header files, or (2) creation of a new configuration module which supplies external definitions and/or call-backs to variant components at run-time. In either case, one resorts to the use of C pre-processor at compile-time, or uses a C-based Application Programming Interface (API) calls at run-time for configuration.

The C (pre-processor), however, offers limited options for calculation of configuration parameters at compile-time, a notable omission being recursive macros. In addition, manual (re-) configuration is error-prone. This frequently leads to postponing of the configuration to run-time (or init-time), complicating start-up procedures and potentially delaying processing at reconfiguration time, which can be significant in SDR.

![Figure 10. Specifying configurations.](image)

In LIME, these shortcomings are addressed via provision of a narrow, C preprocessor-based interface between the variant components (written in C) and the configurator (which uses FP). This interface consists of name-value pairs, possibly annotated using C/C++ comments (see Fig. 10). From this simple description, a header file is automatically generated and subsequently included by slimer, piggy-backing on the compilation-domain feature.

3.2.3 Data-type diversity

A frequent cause for using `#include` directives is to import configurations, data-type definitions, associated dependent types, accessor methods, and other function prototypes or macros using these data-types. To rephrase, in standard C, explicit `#include` results in a set of `implicit declarations`.

All structured data-types in C have one of the following type tags: (1) `struct`, (2) `union` or (3) `enum`. It is possible to forward-declare incomplete types by referencing one of the above type tags and a type identifier. Although the definition of types does not cover all potential uses of...
In contrast to prior art, LIME components state such dependencies explicitly, via data-type forward declarations. These express that fact that the code that follows uses an external data-type defined by the compilation environment, typically, local to a group of components participating in communication/synchronization.

Summarizing, in our framework, we have reversed the logic: an explicit data-type forward declaration results in an implicit #include, again, piggy-backing on the compilation domain feature. This brings several advantages. One is that slimer can portably collect all needed headers, given a list of all headers used by the project, as is shown on the left-hand side of Fig. 1. The other is that it can provide them all in one step as a compilation-domain feature (see above). Thus a typical problem of maintaining correct #include chains is automated. Another advantage is that the kernel code is not cluttered with platform-specific references to concrete files, but only needs to specify used data-types, via using “directive” on the right-hand side of Fig. 12 for an example.

3.2.4 Algorithm diversity

Even with all other types of diversity addressed, we have found applications that require actors that use different kernel algorithms depending on platform peculiarities. Such actors typically, but not necessarily, provide the same interface (as a set of ports), but do need to provide very different implementation behaviors. Although this can be accomplished by possibly reusing an interface between different prototypes, we have decided to extend the SDE with a possibility of cloning a prototype (see the components “directive” with .c=.c syntax on the left-hand side of Fig. 12).

This allows the designer to maintain all variations in one module. Although the module is reused physically (see src1.c in Fig. 12) it is still possible to distinguish between variants at fine level of granularity by providing different configurations for each cloned node: src1.c and src2.c can be separately configured.

3.3 Code & Document generation

A pilot SDE implementing our LIME PPM has been developed at NXP and is operational. On the left-hand side of Fig. 13 we show a snapshot of a browser page that was generated automatically using a supplied XML Stylesheet Transformation (XSLT). The source is the GXF model of a realistic SDR application. This model was a supplied XML Stylesheet Transformation (XSLT). The source is a snapshot of a browser page that was generated automatically using NXP and is operational. On the left-hand side of Fig. 13 we show a fish-eye view of a node ordering, as shown on the left-hand side of Fig. 13, on top of node summary tables.

Figure 12. Specifying clones.

4. Conclusions, future work

In this experience report we have tried to address some important software language engineering challenges in creating correct-by-construction SDR solutions. Because DF is an inherent feature of our domain, the meta-model we have applied is rooted in DF. Despite its formal background, we show a number of useful high-level languages that build on top of this meta-model. These are inherently platform-independent: we contribute a number of diversity mechanisms to allow platform-specific tuning. We believe that “independence-by-default” as well as the ability to “optimize-as-needed” for each platform specifically are both essential factors for gaining acceptance of formal modeling in embedded software industry in general, and for SDR in particular.

The desire to support various DSLs in our domain is constrained by the costs in developing the corresponding tooling. We demonstrate that a promising direction in addressing this issue lies in maximizing the extent of reuse of already stable solutions, like the Graphviz, HTML/XML and FP.

Other important issues include decomposability of the tool-flow and compositionality of final models and binaries. For the former, we contribute a flexible XML schema that admits a number of useful simplifications. This schema is shared between bidirectional model¬→-model transformations. For the latter, we introduce a novel technique of transparent modeling, which facilitates 3rd party component development, configuration and integration. The benefits and the pitfalls of using such an XML schema are commonly termed as “fragile” interfaces. Any minor change in the semantics of the schema would require many changes to various transformation tools. A promising direction towards mitigating this danger lies in automation of model¬→-model transformation tool generation.

The final step of our mapping flow involves an OS-specific model¬→source back-end. We have used it to generate code from the final model, link the compiled kernels, and run it on a SDR platform. In our previous work [16] we have validated this strategy by showing that it can result in savings of ≈ 40% in software complexity (#lines, #statements & #cyclomatic [21] metrics) for our DAB application with no measured decrease in run-time performance. Although this result is promising, a conclusive evaluation requires many more experiments with real-life SDR applications.

Finally, seamless document integration is illustrated in this paper. The C code for all actors together with graphs can be taken “as-is” towards working (test) applications, according to literate programming [13] methodology. Although it remains to be seen
how this can be scaled to larger SDR designs, we think that this 
esthetic aspect is in the end at least as important as other aspects (such as efficiency, dependability and cost) in achieving the aim of correct-by-construction SDR.

4.1 Future work

One of the interesting directions for development of the work reported here includes bringing proposed representation of the streaming aspect in DF as a centerpiece of the Web-based development environment for SDR applications. We plan to use Hop [24], a multi-tier, 


diffuse Web 2.0 solution which is also based on FP & Bigloo. In addition, we would like to evaluate our approach on larger real-life SDR designs from the automotive domain.

References

### A. Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOP</td>
<td>Aspect Oriented Programming</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ARM</td>
<td>Acorn RISC Machine (or Advanced RISC Machine)</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific IC</td>
</tr>
<tr>
<td>ATL</td>
<td>Atlas Transformation Language</td>
</tr>
<tr>
<td>AWK</td>
<td>Aho, Weinberger, Kernighan (text processor)</td>
</tr>
<tr>
<td>BDF</td>
<td>Boolean DF</td>
</tr>
<tr>
<td>BE</td>
<td>Back-End</td>
</tr>
<tr>
<td>CBSE</td>
<td>Component-Based SW Engineering</td>
</tr>
<tr>
<td>CLI</td>
<td>Command Line Interface</td>
</tr>
<tr>
<td>CSDF</td>
<td>Cyclo-Static DF</td>
</tr>
<tr>
<td>DAB</td>
<td>Digital Audio Broadcast</td>
</tr>
<tr>
<td>DF</td>
<td>Data-Flow</td>
</tr>
<tr>
<td>DOM</td>
<td>Document Object Model</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain-Specific Language</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>DTD</td>
<td>Document Type Definition</td>
</tr>
<tr>
<td>DVB</td>
<td>Digital Video Broadcast</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modeling Framework</td>
</tr>
<tr>
<td>FE</td>
<td>Front-End</td>
</tr>
<tr>
<td>FFI</td>
<td>Foreign Function Interface</td>
</tr>
<tr>
<td>FIFO</td>
<td>First-In First-Out</td>
</tr>
<tr>
<td>FP</td>
<td>Functional Programming</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>GLPK</td>
<td>GNU Linear Programming Kit</td>
</tr>
<tr>
<td>GXF</td>
<td>Graph Exchange Format</td>
</tr>
<tr>
<td>HTML</td>
<td>Hyper-Text Modeling Language</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>IO</td>
<td>Input-Output</td>
</tr>
<tr>
<td>LIME</td>
<td>Less Is More</td>
</tr>
<tr>
<td>MDSd</td>
<td>Model-Driven Software Development</td>
</tr>
<tr>
<td>MIC</td>
<td>Model-Integrated Computing</td>
</tr>
<tr>
<td>MoC</td>
<td>Model of Computation</td>
</tr>
<tr>
<td>NXP</td>
<td>Next Experience Semiconductors</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PIM</td>
<td>Platform Independent Model</td>
</tr>
<tr>
<td>PPM</td>
<td>Parallel Programming Model</td>
</tr>
<tr>
<td>PSM</td>
<td>Platform Specific Model</td>
</tr>
<tr>
<td>QVT</td>
<td>Query/View/Transformation</td>
</tr>
<tr>
<td>R5RS</td>
<td>Revised(^5) Report on Algorithmic Language Scheme</td>
</tr>
<tr>
<td>SAX</td>
<td>Simple API for XML</td>
</tr>
<tr>
<td>SDE</td>
<td>Software Development Environment</td>
</tr>
<tr>
<td>SDF</td>
<td>Synchronous DF</td>
</tr>
<tr>
<td>SDR</td>
<td>Software-Defined Radio</td>
</tr>
<tr>
<td>SSA</td>
<td>Static Single Assignment</td>
</tr>
<tr>
<td>SSAX</td>
<td>Scheme SAX</td>
</tr>
<tr>
<td>SXML</td>
<td>Scheme XML</td>
</tr>
<tr>
<td>TUD</td>
<td>TU Delft</td>
</tr>
<tr>
<td>UML</td>
<td>Universal Modeling Language</td>
</tr>
<tr>
<td>VCS</td>
<td>Version Control System</td>
</tr>
<tr>
<td>WLAN</td>
<td>Wireless LAN</td>
</tr>
</tbody>
</table>