Supporting Heterogeneous Architecture Descriptions in an Extensible Toolset

Matthieu Leclercq, Ali Erdem Özcan
STMicroelectronics
Grenoble, France

Vivien Quéma
CNRS
Grenoble, France

Jean-Bernard Stefani
INRIA
Grenoble, France

Abstract

Many architecture description languages (ADLs) have been proposed to model, analyze, configure, and deploy complex software systems. To face this diversity, extensible ADLs (or ADL interchange formats) have been proposed. These ADLs provide linguistic support for integrating various architectural aspects within the same description. Nevertheless, they do not support extensibility at the tool level, i.e. they do not provide an extensible toolset for processing ADL descriptions.

In this paper, we present an extensible toolset for easing the development of architecture-based software systems. This toolset is not bound to a specific ADL, but rather uses a grammar description mechanism to accept various input languages, e.g. ADLs, Interface Definition Languages (IDLs), Domain Specific Languages (DSLs). Moreover, it can easily be extended to implement many different features, such as behavioral analysis, code generation, deployment, etc. Its extensibility is obtained by designing its core functionalities using fine-grained components that implement flexible design patterns.

Experiments are presented to illustrate both the functionalities implemented by the toolset and the way it can be extended.

1 Introduction

Problem statement As defined in [14], the architecture of a software system refers to its organizational structure. This architecture can either be implicit or explicit. Making the architecture explicit has been an active area of research aiming at easing and improving the development process of software systems. These works led to the development of various Architecture Description Languages (ADL) [19]. Besides describing the architecture of a software component, ADLs allow specifying various aspects such as its behavior [16], its interaction protocols [4], and platform-dependent information [10, 20]. Various tools have been developed to analyze, configure, or deploy these descriptions. Unfortunately, these tools focus on a specific purpose and are not easily extendable to support other functionalities. In particular, they fail to provide support for what we would call heterogeneous architecture descriptions, where the description of the software architecture involves multiple languages: ADLs, Interface Definition Languages (IDLs), Domain Specific Languages (DSLs), etc. Providing an integrated support for heterogeneous architecture description is key to easing the development of complex systems. Attempts have been made to develop extensible ADLs. For instance, ACME [13] proposes an extensible interchange format to federate various ADLs. In the same context, xADL [10] defines an extensible ADL that allows uniformly describing many aspects of the application architecture. However, both propositions focus only on the extensibility of the architecture description language itself, and do not provide an associated extensible toolset to process heterogeneous architecture descriptions. The problem we address in this paper is precisely that of devising an extensible toolset for processing heterogeneous architecture descriptions.

Motivating scenario We illustrate the notion of heterogeneous architecture descriptions with an example of real-world development (Figure 1). Consider a multimedia application that integrates a GUI and two video decoders. One of the decoders and the GUI are legacy code (C and Java, respectively), whereas the second decoder has a component-based structure, whose description involves the use of a DSL (*JSL* for *Join Specification Language*) for describing synchronization and concurrency at the architecture level[1]. The interfaces of the two decoders are described using an IDL (*FractalIDL*). Finally, an ADL (*FractalADL*) is used to describe the interfaces of the GUI and overall architecture (i.e. component instances and their connections).

What we expect from a toolset supporting such an heterogeneous architecture description is that it should com-

[1] *JSL* is a language based on the join calculus [12] that allows describing event-condition-reaction (ECA) rules. Basically, join patterns are used to describe conditions using patterns of events. Once a pattern is matched, the associated reaction (implemented by a component) is triggered.

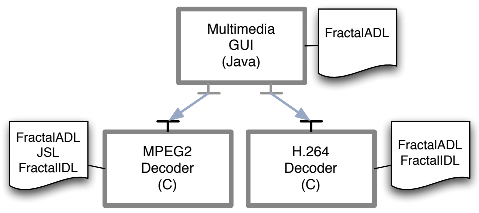

Figure 1. Motivating scenario.

bine multiple features including verification of the architecture's correctness (e.g. type checking component assemblages as described in [6]), generation of glue code for Java and C components, generation of JNI channels to allow cross-platform communications, generation of synchronization protocols implied by the *JSL* descriptions, and deployment of the architecture on a target platform (according to the *FractalADL* description).

Contributions The main contribution of our work is the definition of an extensible framework for heterogeneous architecture description processing (including architectural analysis, code generation, deployment, etc.). This is achieved thanks to a component-based design (using the Fractal model [8]) and the combination of several design patterns with inherent extensibility. Among the benefits of our work, we can emphasize:

- the ability to support an extensible ADL. This is illustrated by the support of the extensible Fractal ADL, and we could as easily envisage the support of xADL, for instance.
- the ability to support multiple targets, in terms of programming languages, as well as execution platforms.
- the ability to integrate, in a common toolset, multiple specialized tools covering different architectural concerns (e.g. analysis, deployment, etc.).
- the ability to automate the generation of code implementing architecture elements implied by the architecture description (e.g. connectors for cross-platform communications or for distributed execution).

Outline This paper is organized as follows. Section 2 presents an architectural overview of the toolset. Section 3 presents the tree structure (AST) that is used to model systems described using heterogeneous architecture descriptions. Section 4 and 5 describe the architectural and programing patterns that are used for the construction and the processing of the AST. In each of the last three sections, extensions made to the described components for supporting the motivating scenario are discussed. Section 6 explains how the previously described extensions can be combined to deploy the motivating scenario. Section 7 discusses related work. Section 8 concludes the paper.

2 Architecture overview

Figure 2 presents the workflow implemented by the toolset. The toolset comprises three main components that we briefly describe below.

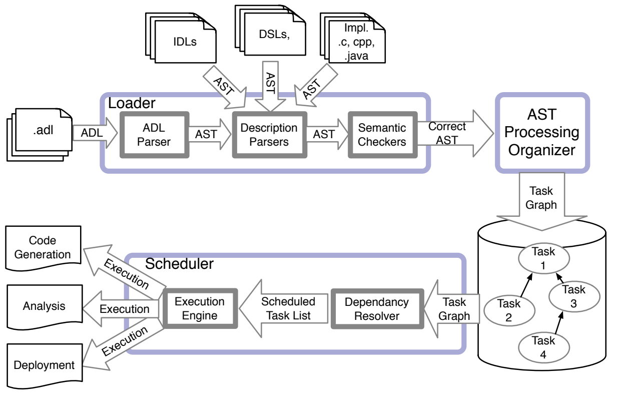

Figure 2. Toolset workflow.

The *loader* component reads a set of input files (e.g. ADL, IDL, DSL, etc.) and produces an Abstract Syntactic Tree (AST). This tree provides a unified representation of the system architecture that is described using various description languages. The AST is generated using grammars that abstract each input language. Note that other unified representation formats may have been chosen, e.g. plain XML documents. Nevertheless, it is more efficient to process an AST than an XML document. This comes from the fact that processing XML documents requires manipulating files. The *loader* also integrates an extensible set of semantic analysis functionalities (e.g. assembly correctness, behavioral compatibility, etc.). Note that the semantic analysis may modify the tree. For instance, we describe later in the paper how the *loader* can be extended to insert JNI channels between cross-platform components.

The *organizer* component processes the AST to generate a graph of tasks to be executed. Examples of tasks include generation of code for JNI channels, instantiation of components, deployment of connectors between components, etc. The task graph allows specifying both the dependencies between tasks and the data flow they exchange. For instance, the instantiation of a component may provide a name that will be used by another task to configure a connector.

The *scheduler* component schedules the execution of each task of the graph in an order that guarantees that dependencies and data flow are respected.

3 The Abstract Syntactic Tree (AST)

This section describes the AST architecture and the factory mechanism that is used to generate it from abstract grammar descriptions. The extensibility of the AST is illustrated through examples involving the application presented in the introduction (Figure 1).

3.1 AST architecture

The AST is implemented as an XML-based object tree. Each node of this tree has multiple interfaces. First, each node inherits a base class providing generic functions for adding and removing decorations. These decorations are used by the various components processing the AST. For instance, a module in charge of checking the correctness of the system architecture may add a decoration to a node representing a required interface in order to note whether it is bound to another interface or not. In addition, each node implements an interface that is specific to the type of the node. This interface gives access to a set of arguments. For instance, the type-specific interface of a node representing a component may give access to the component's name and to the location of its source code. Finally, a node can have children. In such case, it is said to be a *container* node and it implements one or several container interfaces to give access to its children. For example, a node representing a component can have children representing its interfaces and its subcomponents[2]. In this case, it will have two container interfaces, for the interfaces and subcomponents, respectively.

The fact that each node can have many interfaces (i.e. many container interfaces), makes the AST inherently extensible. For instance, taking into account a new input language can most of the time be done by adding a new container interface acting as a merge point between the two languages. Consider the AST depicted in Figure 3. This AST represents part of the application described in the introduction. The white nodes are generated from the *FractalADL* description of the GUI component. The root node represents the GUI. It implements the `Component` interface that gives access to the name and to the source file of the component. The *GUI* component has two client interfaces (for H.264 and MPEG2 streams) that are represented by two children nodes. These children nodes can be accessed using the `InterfaceContainer` interface. Interfaces of the GUI component are described using the *FractalIDL* language. These descriptions are used to generate the grey part of the AST. This part models the methods and fields of one of the interfaces (the H.264 interface). As we can see, the integration of *FractalIDL* descriptions is done by adding an `InterfaceDefinitionContainer` interface to the nodes representing the interfaces. Consequently, supporting a new language does not require any modification to the interfaces of existing nodes.

[2]In hierarchical models, components may contain sub-components.

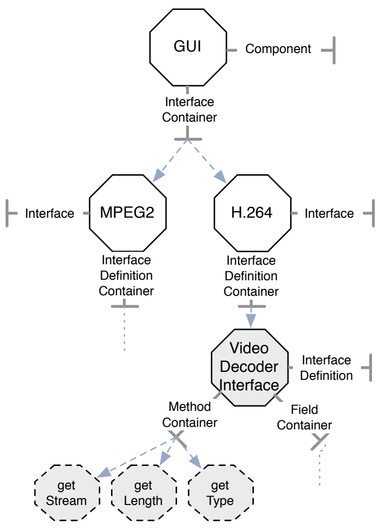

Figure 3. Example of an AST.

3.2 AST factory

As we have seen in the previous section, the AST can be modified to integrate new input languages. It is obviously not reasonable to have an hand-coded AST implementation. Consequently, we use an AST factory that provides an API to create nodes of the AST. The types of nodes that can be created by the factory are described using a basic grammar specification language. This language is based on XML/DTD.

Let us illustrate the grammar specification language using the AST depicted in Figure 3. The AST factory must allow the creation of nodes representing components and interfaces as described in *FractalADL* (white part of the tree). The grammar specification is the following:

```
 1. <?add ast="component" itf="adl.Component"?>
 2. <!ATTLIST component
 3.     name CDATA #REQUIRED
 4.     source CDATA #IMPLIED>

 5. <?add ast="component"
 6.         itf="adl.InterfaceContainer"?>
 7. <!ELEMENT component (interface*) >

 8. <?add ast="interface" itf="adl.Interface"?>
 9. <!ATTLIST interface
10.     name CDATA #REQUIRED
11.     signature CDATA #REQUIRED>

12. <!ELEMENT interface () >
```

Lines 1 and 8 specify that there exist two kinds of nodes in the AST, which have for type-specific interface the `Component` and `Interface` interfaces, respectively. Lines 2 to 4 and 9 to 11 specify the arguments that can be accessed using these two type-specific interfaces. Line 7 specifies that a node representing a component may have multiple interfaces that can be accessed through the `InterfaceContainer` interface (Lines 5 and 6). Finally, Line 12 describes the fact that an interface node does not have any child node.

In order to extend the AST with information coming from *FractalIDL* definitions (grey part), it is first necessary to write a new grammar description similar to the one presented for the *FractalADL* language. Then, the grammar of the `FractalADL` language must be extended in order to define a container interface that will allow accessing nodes representing interface definitions. This is done by replacing Line 12 with the two following lines (`itfDef` refers to the grammar of the *FractalIDL* language):

```
12. <?add  ast="interface"
13.        itf="adl.ItfDefContainer"?>
14. <!ELEMENT interface (ItfDef?) >
```

4 Building a correct AST

This section describes the *loader* component that creates an AST using descriptions in heterogeneous architecture description languages. We start by describing the architecture of the *loader*. Then, we present the syntactic and semantic analysis modules that it integrates. We also discuss its extensibility using the example of the *JSL* language.

4.1 Loader architecture

The *loader* is composed by a chain of components implementing a recursive loading process, where the base of the recursion is implemented by a low-level parser component corresponding to the main input language (Figure 4). In the example described in the introduction (Figure 1), the main input language is *FractalADL*. Indeed, this language is used to model the overall architecture.

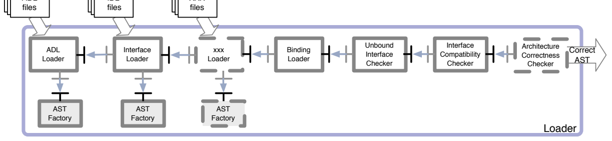

Figure 4. Architecture of the *loader* chain.

Using a chain as the design pattern for the *loader* component allows a hierarchical processing of errors. Indeed, each component receives a correct AST from the component that immediately follows it in the chain. When an error is detected, the component throws an exception, which will be handled by all the components preceding it in the chain. The only issue to consider when adding a new component in the chain is to find an adequate location. Indeed, the component must be inserted before a component which will return an AST on which the expected verifications will have been done. For instance, the *unbound interface checker* component depicted in Figure 4 requires all component interfaces and their connectors to be loaded. Consequently, this component must precede the *binding loader* component.

We distinguish two kinds of components in the chain: *parser* components participate to the construction of the AST using one input file; *semantic analyzer* components proceed to semantic verifications on the AST they receive. Note that these components may also modify the AST if necessary.

4.2 Parsers

As explained before, parsers contribute to the AST construction. Each parser is associated to an input language. It first proceeds to a syntactical analysis of the description it processes. Then, it uses the AST factory described in Section 3.2 to create AST nodes mapping the described architecture. We have developed several parsers. For instance, we have developed a generic parser that can be used when the input language is based on XML. This generic parser is customized using the grammar written for the AST factory. When the input language is not based on XML (e.g. *FractalIDL* and *JSL* in our example), it is necessary to develop a specific parser. Our experience shows that this is a straightforward engineering task since many existing tools provide helpful support (e.g. *yacc*, *javacc*).

4.3 Semantic analyzers

Semantic analyzers perform verifications on the generated AST. A typical example of verification consists in checking that the architecture to be deployed is correct (e.g. components are connected using compatible interfaces). Analyzers can also modify the AST to enhance or to optimize the modeled architecture. Figure 5 depicts part of the architecture of the motivating scenario presented in the introduction. A Java-based GUI component is connected to a C-based H.264 video decoder. We implemented an AST analyzer, whose role is to insert JNI channels to allow components written in C and Java to interact. Such channels are made of a pair of `Stub`/`Skeleton` components.

The modification performed by the analyzer component on the AST is depicted in Figure 6. Before the transformation, the `MMPlayer` node contains three children repre-

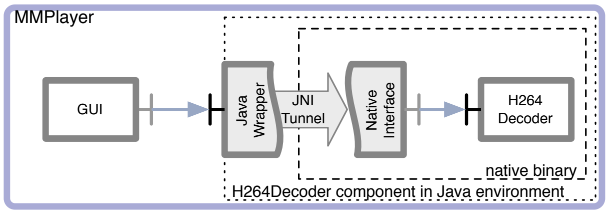

Figure 5. An architecture with C- and Java-based components.

senting the GUI and H.264 components, as well as the connector between the latter components. In the transformed AST, the connector node is replaced by a `Stub`/`Skeleton` pair, as well as connectors from the GUI (resp. H.264) component to the `Stub` (resp. `Skeleton`) component. Note that we used a gray color for the `Stub` and `Skeleton` nodes to represent the fact that they have a decoration specifying that their implementation must be generated.

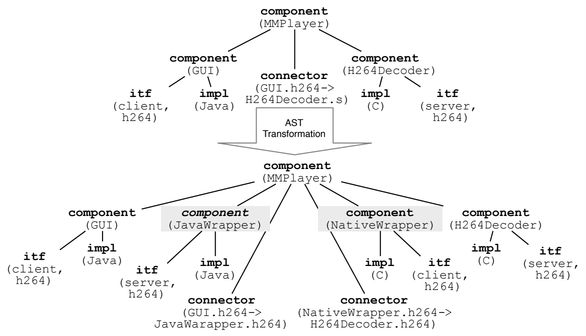

Figure 6. AST transformation for inserting JNI wrappers between C- and Java-based components.

4.4 Extending the *loader* for *JSL*

This section explains how the *loader* has been extended to support the *JSL* language used in the example presented in Figure 1. As depicted in Figure 7, several components have been inserted in the *loader* chain. Insertion is made before the *interface loader* component because the added *loader* modules act on components' interfaces. The *JSL loader* module travels the AST and detects all the components whose behavior is described using a *JSL* description. It uses the *join parser* module to create the ASTs corresponding to each of these descriptions. Then, the *join decorator* merges these ASTs with the overall AST. Finally, the *join semantic checker* module verifies that components have a behavior compatible with their *JSL* description.

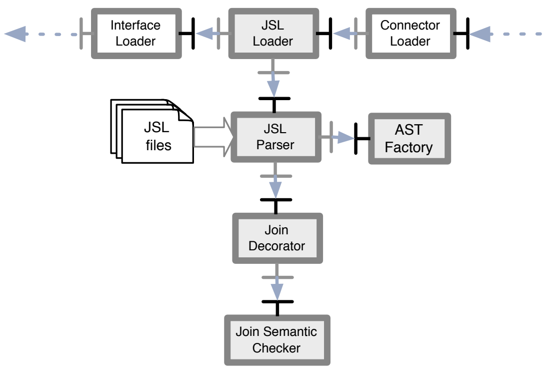

Figure 7. Integration of additional components in the *loader* chain to support *JSL*.

5 Processing the AST

There is plethora of tools that can be developed for easing the architecture-based development process. Let us cite, for instance, tools for behavioral analysis, code generation, deployment, etc. In this section, we describe a framework that allows developing such tools based on the correct AST that is generated by the *loader* component. As explained in Section 2, our AST processing framework relies on the use of an extensible task graph. This graph is built by *organizer* components and executed by a *scheduler* component. We first describe the task graph architecture; then, we briefly present the schedulers, before presenting the design patterns that *organizer* components implement; finally, we show how the AST processing framework can be extended.

5.1 Task graph architecture

The *task graph* represents a set of tasks, together with their dependencies and the flow of data they exchange. The task graph must be acyclic in order to guarantee that the *scheduler* will be able to execute it. Each task implements a specific functionality. Typical tasks include data structure generation, component instantiation, connector deployment, etc. Tasks may receive inputs from two different sources: the AST and other tasks. This allows tasks to collaborate.

An example of collaboration between tasks is depicted in Figure 8. Dependencies and data flows between tasks are expressed using standard grammar rules. The left part of a rule represents a task; the right part describes its result. Verbatim words are integrated as such in the generated code, while underlined words are replaced by information stored in the AST. The example that is shown illustrates part of the tasks that are used to generate glue code for the GUI

CompDef	→	`public abstract class` CompType `implements` ProvServ_1 {, ProvServ_i*} { ReqServ* }
ProvServ	→	ProvServType
ReqServ	→	ReqServType ReqServName ;

Figure 8. Collaboration model for a set of tasks generating components in Java.

component. This component is written in Java. Generating the glue code consists in defining a class that implements a set of provided services and that contains fields representing required services. Three tasks are used. A main task, called `CompDef`, writes the code of the class. It starts by writing usual keywords (e.g. `public abstract class`) and then uses information stored in the AST (underlined words) as well as results of two other tasks that return the code for provided and required service declarations, respectively. Note that the only input of the two latter tasks is data stored in the AST.

We have defined a simple API for creating and manipulating task graphs. Basically, this API provides methods for (un)registering tasks, for retrieving already registered tasks, and for defining data exchange and dependency rules between tasks.

5.2 Schedulers

Task graphs are executed by *scheduler* components. A task can be executed as soon as its dependencies are satisfied. Consequently, the first tasks to be executed are the ones that only use information stored in the AST (`ProvServ` and `ReqServ` in the previous example). The *scheduler* can implement various parallelization strategies for the execution of tasks. Presenting them is out of the scope of this paper.

5.3 Organizers

The task graph is generated by the *organizer* component using the AST as input. The *organizer* architecture follows the *Visitor* pattern. We distinguish two types of *organizer* components:

- *AST traveler* components travel through the AST. They use a set of *Organization visitor* components that they invoke each time they find a node representing a component.
- *Organization visitor* components are invoked by the AST traveler components. They only process the subpart of the AST that has for root the node they currently visit. Their role is to register a set of tasks to be executed, together with their dependencies and data flow constraints.

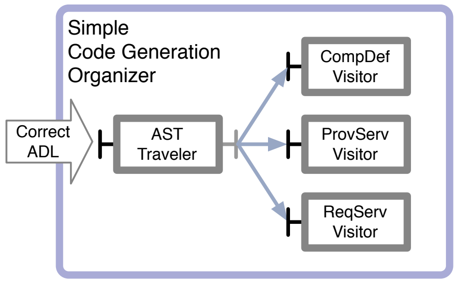

Figure 9. Architecture of a simple code generator.

Let us illustrate how this design pattern leads to the generation of the task graph presented in Section 5.1. The architecture of the *organizer* component in charge of generating code for Java components is depicted in Figure 9. Besides the *AST traveler*, the *organizer* component contains three *organization visitors* that are responsible for the three tasks required to generate Java code. For instance, the *component definition organizer* creates a new `CompDef` task each time it visits an AST node representing a component. This task is initialized with the component type name (CompType) stored in the AST. Similarly, the *provided services organizer* component creates a new task each time a node representing a provided service is found in the AST. This task is registered as a source code producer for the `CompDef` task created for the component that provides the visited service.

As explained before, *organizer* components implement the *Visitor* pattern. Consequently, they all have the same code organization, which eases their implementation. Figure 10 depicts the pseudo-code of the *component definition organizer*. The `visit` method implemented by the *organizer* has for parameter the AST node representing the visited component and the task graph. The *organizer* first creates a `CompDef` task and initializes it with the type information stored in the AST. It then searches in the task graph the `ProvServ` tasks associated to the provided services of the node it visits. It declares dependencies between the `CompDef` task and the `ProvServ` tasks. Actually, the `SrcCodeProducer` parameter specifies that the code generated by `ProvServ` tasks will be used by the `CompDef` task. It then does the same processing for `ReqServ` tasks. Finally, it registers the `CompDef` task.

```
procedure compDefVisit(compNode, taskGraph) {
 // Create a CompDef task
 compDefTask = new CompDefTask() ;

 // Initialize the CompDef task
 compDefTask.setCompType(compNode.type);

 // Find and add dependencies to ProvServ tasks
 ProvServ[] provServs = compNode.provServs;

 foreach (provServ in provServs) {
  provServTask = taskGraph.getTask(provServ.name,
                                   compNode) ;
  compDefTask.addDependency(provServTask,
                            SrcCodeProducer) ;
 }

 // Same for ReqServ tasks
 ...

 // Register the CompDef task
 taskGraph.registerTask('CompDef',
                        compNode,
                        compDefTask);
}
```

Figure 10. Pseudo-code of the *component definition organizer* **component.**

5.4 Extending the AST processing framework

In this section, we describe two extensions to the AST processing framework. The first extension shows how new tasks can be defined and inserted in the task framework in order to generate code for component type definitions. The second extension shows how code can be generated for supporting the *JSL* language.

5.4.1 Code generation for component type definitions

Consider that a system architect wants to use the AST processing framework to generate and compile component type definitions conforming to the rules depicted in Figure 8. He will first have to define new types of tasks[3]. Examples of such task types are depicted in Table 1. Data produced (resp. consumed) by tasks are represented by up-arrows (resp. down-arrows). Note that for the sake of clarity, only data flows between tasks are presented, i.e. data retrieved from the AST are not shown. The `SrcCodeProducer` interface defines a type of tasks that produce source code, while the `SrcCodeConsumerProducer` interface defines a type of tasks that both produce and consume source code. Similarly, the `SrcFileProducer` and `FileConsumerProducer` interfaces define types of tasks that produce files.

[3]The notion of type of tasks is necessary for handling the dependencies between tasks. Indeed, the `addDependency` method (see Figure 10) takes this type as parameter in order to constrain the flows of data that are exchanged between interdependent tasks.

	source code	file
`SrcCodeProducer`	↑	-
`SrcCodeConsumerProducer`	↓ ↑	-
`SrcFileProducer`	↓	↑
`FileConsumerProducer`	-	↓ ↑

Table 1. Types of tasks for code generation and compilation.

Once task types have been defined, the system architect must provide actual implementations of them. For instance, the `ProvServ` and `ReqServ` tasks introduced in the previous sections are implementations of the `SrcCodeProducer` type. Another example is the `CompDef` task that is an implementation of the `SrcCodeConsumerProducer` type. As illustrated in Figure 10, these actual implementations are then used in the code of *organizer* components. Developing these components and integrating them in the AST processing framework is easily done using the *Visitor* pattern described in the previous section.

5.4.2 Code generation for the *JSL* language

As explained in the introduction, *JSL* is a language based on the join calculus [12] that allows describing event-condition-reaction (ECA) rules. In this section, we show how the AST processing framework can be extended to generate a C-based implementation of finite state machines (FSM) managing ECA rules. Similarly to the architecture presented in [11], FSMs are implemented using three main modules (see Figure 11). `MessageQueues` allow storing received messages from a given type. Obviously, the FSM of a component can contain multiple `MessageQueues`. The `QueueManager` provides an API to manage the queues of the FSM. Each time a message is enqueued in one of the queues it manages, the `QueueManager` invokes the `PatternMatcher` to check whether a pattern is matched by messages stored in the `MessageQueues`. When a pattern is matched, the `QueueManager` dequeues the corresponding messages and triggers the reaction of the component.

FSMs are generated by three different organization visitors that create tasks of above-mentioned types. The first visitor is dedicated to the generation of `MessageQueues`. This visitor creates `SrcCodeProducer` tasks that retrieve from the AST the types of messages exchanged by a component and generate code for appropriate data structures. The second visitor cre-

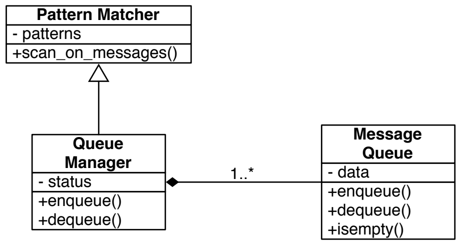

Figure 11. UML diagram of the finite state machines generated for the *JSL* language.

ates `SrcCodeConsumerProducer` tasks that generate source code for the `PatternMatcher`. The latter tasks consume the code generated by tasks generating `MessageQueues` and retrieve patterns to be matched from the AST. Finally, the third visitor aims at generating the `QueueManager`'s implementation. It creates `SrcCodeConsumerProducer` tasks that generate code using both the code generated for the `MessageQueues` and the `PatternMatcher`.

6 Back to the motivating scenario

In this section, we explain how the various features presented in the paper allow deploying our motivating, real-world example depicted in Figure 1.

1. **Verification of the architecture's correctness:** as presented in Section 4, our tool comprises a loader chain in charge of generating an AST representing the overall architecture of the application to be deployed. The correctness of the architecture is assessed by loaders that check that components are correctly bound (i.e. that all interfaces are bound to interfaces with compatible types). Note that complex type-checking analysis, such as the one presented in [6], can be performed.

2. **Generation of glue code for components:** we have presented in Section 5 the way the AST can be processed. In particular, we illustrated the AST processing with the example of glue code generation for components written in Java (Section 5.4.1). Similar code generation tasks have been developed for generating the glue code for C-based components. In our motivating example, they are used to generate data structures for the MPEG2 and H.264 decoders.

3. **Generation of JNI channels:** our motivating example involves components written in different languages (C and Java). We have shown in Section 4.3 that the *loader* component could be extended to automatically add JNI stub/skeleton components between components. The code of these stub/skeleton components is automatically generated by tasks similar to the ones presented in Section 5.4.1.

4. **Support for the *JSL* language:** the MPEG2 decoder uses complex synchronization primitives described in the *JSL* language. We have shown in Section 4.4 that the loader chain can be extended for integrating *JSL* descriptions in the AST and for performing behavioral verifications. Moreover, we have explained in Section how the AST processing framework can be extended to allow generating the implementations of finite state machines required for managing components' execution.

5. **Compilation of sources:** generated and hand-coded sources are compiled to produce executable binaries. Compilation tasks have not been presented in the paper. These tasks are basic since they rely on standard compilers (e.g. gcc, ld, javac).

6. **Instantiation of components:** executable binaries are instantiated by dedicated tasks. For space limitations, these tasks have not been presented. Basically, they work as follows: Java components (including JNI wrappers) are directly deployed on a target JVM. C components are first packaged as shared libraries. They are then loaded by the JNI stub/skeleton components as soon as a connection has to be established between the components.

7 Related Work

There have been several ADL developments in the past fifteen years, which can broadly be classified in two categories according to the support tools they provide. A first category includes ADLs such as Wright [4], Rapide [16] or Unicon [23] that provide assistance for the design and analysis of software systems. They allow system designers to specify the dynamic behavior of the modelled components, by means of different formalisms, including process calculi, and provide tools to help verify various safety and liveness properties. A second category includes ADLs that provide assistance for the actual implementation of software systems. This is the case of ADLs such as Olan [5], Aster [15], C2 [18], Darwin [17], or Knit [21] that use architecture descriptions to automate software configuration and deployment, connecting architecture descriptions and the underlying programming language in which the system is implemented. Interesting cases in this family are the ArchJava [2] and the ComponentJ [22] programming languages, which actually merge architecture description with programming, extending the Java language with component constructions. ArchJava proposes an interesting language support for developing software connectors implementing various communication semantics [3]. Our toolset provides a similar support as illustrated with the JNI example. Note that our proposition differs from the former in the ability to automate the code generation and deployment of connector components using architectural information (e.g. deployment of JNI channels between Java- and C-based components).

The above ADLs have in common to be somehow monolithic, even though they may provide a certain degree of extensibility: for instance, this is the case of Knit, which allows a system designer to extend the Knit tool chain to take into account additional design constraints to be verified against architecture descriptions. Some recent works, however, recognize that the many different architectural concerns (typically addressed by different ADLs) need to be supported. This line of thought underpins the proposition of ACME [13], an architecture description interchange language between different ADLs, but also, more importantly, recent proposals for extensible ADLs such as ADML [1], xACME [24], and xADL [10]. ADML is a translation of ACME into an XML DTD. xADL and xACME are based on XML-schemas and built above xArch [10], an XML-based infrastructure for the development of ADLs. xADL introduces an interesting distinction between *architecture description* and *architecture prescription*. The former relates to a system structure at runtime, whereas the latter relates to a system structure at design time, before its instantiation. The focus in xADL is on the prescription side, with particular support for architecture evolution and product families, representing concepts such as options and variants, and more generally configuration management with versions. xADL also provides modules for abstract and Java implementation. xACME extends xArch with constructs for specifying properties and constraints on architecture families.

Finally, our work has some relationship with model-driven engineering approaches and work around the Unified Modeling Language [7]. UML 2 comprises support for architecture descriptions. Several tools, from commercial CASE tools such as Rational Rose, to more experimental ones such as Fujaba [9], support UML components to some level, and provide extensive model analysis and transformation capabilities, as well as code generation capabilities. Our work is centered on architecture-based inputs, and does not pretend to address all the above mentioned features. However, we believe that by integrating ADL, IDL and XMI (XML Metadata Interchange) inputs, our toolset could be used to implement many features including code and documentation generation, as well as architecture deployment.

To summarize, the toolset presented in this paper has two main differences compared to previous propositions. First, our toolset does not implement a monolithic function. Instead, it provides a modular and extensible core which can integrate any of them. Extensions for code generation, compilation and deployment have been illustrated in the paper. We believe that support for behavioral analysis similar to the one proposed in Rapide and Wright could easily be implemented using the AST processing framework. We plan to implement such features in our future work. The second main difference is that our toolset does not rely on any specific ADL, but rather accepts heterogeneous architecture descriptions in various languages. We argue that the ability to integrate different legacy (domain-specific) languages is key to supporting the development of software systems in different contexts.

8 Conclusion

In this paper, we have presented an extensible toolset aiming at easing architecture-based software development. The key contributions of our toolset are that it supports heterogeneous architecture descriptions in various languages, and can produce different, extensible forms of outputs. This is achieved using an extensible AST architecture, together with a programmable, component-based processing chain. We have illustrated the extensibility of the proposed toolset with several examples: support of a new input language, insertion of semantic analysis features, code generation for various programming languages, deployment of software components on various execution platforms, etc.

We are currently extending this toolset to perform advanced type-checking analysis [6]. Moreover, it is currently used within many research projects focusing on the development of customized operating system kernels, distributed multimedia applications, configurable communication middleware, etc.

Source accessibility The main source code of the toolset is available at: `http://fractal.objectweb.org`.

Acknowledgments We would like to thank Eric Bruneton, Thierry Coupaye and Philippe Merle for their contributions in the design and the implementation of the toolset. We also thank Frédéric Mayot who has implemented the extensions related to the *JSL* language. We would also like to thank Sacha Krakowiak and the anonymous reviewers for their precious comments on the paper. Finally, we would like to thank Philippe Guillaume for his support and for the technical discussions we had with him.

References

[1] ADML web site. http://www.opengroup.org/architecture/adml/adml_home.htm.

[2] J. Aldrich, C. Chambers, and D. Notkin. Archjava: connecting software architecture to implementation. In *24th ICSE*. ACM Press, 2002.

[3] J. Aldrich, V. Sazawal, C. Chambers, and David Notkin. Language Support for Connector Abstractions. In *Proceedings 17th European Conference on Object-Oriented Programming (ECOOP)*, 2003.

[4] R. Allen, D. Garlan, and R. Douence. Specifying Dynamism in Software Architectures. In *Workshop on Foundations of Component-Based Soft. Eng.*, 1997.

[5] R. Balter, L. Bellissard, F. Boyer, M. Riveill, and J.Y. Vion-Dury. Architecturing and Configuring Distributed Applications with Olan. In *Proceedings IFIP/ACM Middleware Conference*, 1998.

[6] P. Bidinger, M. Leclercq, V. Quéma, A. Schmitt, and J.-B. Stefani. Dream Types - A Domain Specific Type System for Component-Based Message-Oriented Middleware. In *4th Workshop on Specification and Verification of Component-Based Systems (SAVCBS'05)*, Lisbon, Portugal, September 2005.

[7] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1998.

[8] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The Fractal Component Model and its Support in Java. *Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 36(11-12):1257–1284, 2006.

[9] Sven Burmester, Holger Giese, Martin Hirsch, Daniela Schilling, and Matthias Tichy. The Fujaba Real-Time Tool Suite: Model-Driven Development of Safety-Critical, Real-Time Systems. In *Proc. of the 27th ICSE*. ACM Press, 2005.

[10] E. Dashofy, A. Van Der Hoek, and R. Taylor. A Comprehensive Approach for the Development of Modular Software Architecture Description Languages. *ACM Trans. on Software Engineering and Methodology*, 14(2), 2005.

[11] F. Le Fessant and L. Maranget. Compiling join-patterns. In *Proceedings of the Workshop on High-Level Concurrent Languages, Volume 6(3) of Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers*, 1998.

[12] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd Symposium on Principles of Programming Languages*, pages 372–385, New York, NY, USA, 1996. ACM Press.

[13] D. Garlan, R. T. Monroe, and D. Wile. ACME: Architectural Description of Component-Based Systems. In *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[14] IEEE. *Standard Glossary of Software Engineering Terminology*. Std 610.12-1990, 1990.

[15] V. Issarny, C. Bidan, and T. Saridakis. Achieving Middleware Customization in a Configuration-Based Development Environment : Experience with the Aster Prototype. In *Proc. 4th Int. Conf. on Configurable Distributed Systems*, 1998.

[16] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. In *IEEE Trans. on Software Engineering, Vol. 21, No. 4*, 1995.

[17] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *5th ESEC*, volume LNCS 989. Springer-Verlag, 1995.

[18] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. In *Proc. 21st ICSE*, 1999.

[19] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1), 2000.

[20] P. Merle, editor. *CORBA 3.0 New Components Chapters*. OMG TC Document ptc/2001-11-03, November 2001.

[21] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *4th OSDI*, 2000.

[22] J. Costa Seco and L. Caires. A basic model of typed components. In *ECOOP*, volume 1850 of *LNCS*. Springer, 2000.

[23] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *Software Engineering*, 21(4):314–335, 1995.

[24] xACME web site. http://www.cs.cmu.edu/∼acme/pub/xAcme/.