2015 12th Working IEEE/IFIP Conference on Software Architecture

Supporting Dynamic Software Architectures: From Architectural Description to Implementation

Everton Cavalcante[1,2], Thais Batista[1], Flavio Oquendo[2]
[1]*DIMAp, Federal University of Rio Grande do Norte, Natal, Brazil*
[2]*IRISA-UMR CNRS/Université de Bretagne-Sud, Vannes, France*
evertonrsc@ppgsc.ufrn.br, thais@ufrnet.br, flavio.oquendo@irisa.fr

***Abstract*—Dynamic software architectures are those that describe how components and connectors can be created, interconnected, and/or removed during system execution. Most existing architecture description languages (ADLs) provide a limited support to expressively describe these architectures and entail architectural mismatches and inconsistencies between architecture and implementation due to their decoupling from implementation. In this paper, we introduce the dynamic reconfiguration support provided by π-ADL, a formal, well-founded theoretically language for describing dynamic software architectures under structural and behavioral viewpoints. π-ADL provides architectural-level primitives for specifying programmed dynamic reconfigurations, i.e., foreseen changes described at design time and triggered at runtime. In addition, π-ADL allows enacting dynamic reconfiguration by means of: (i) an exogenous approach, in which it is possible to control all elements of the software architectures and to apply the changes on the whole structure; and (ii) an endogenous approach, in which the architectural elements can manage dynamic reconfiguration actions. Furthermore, π-ADL is integrated with the Go programming language, thus enabling to automatically generate implementation code from architectural descriptions, thus tackling the existing gap between them. We hereby use a real-world flood monitoring system as an illustrative example of how to describe dynamic software architectures in π-ADL and automatically generate source code in Go.**

***Keywords*-software architectures; dynamic reconfiguration; architecture description language; π-ADL; Go**

I. Introduction

Dynamism is an important concern for many modern software systems, which often have to undergo modifications for coping with the highly dynamic environments in which they are deployed and/or changing user requirements. As a response to this challenge, *dynamic reconfiguration* was introduced as a mechanism for applying changes on the structure and/or behavior of a system during its execution (i.e., at *runtime*) with minimal or no interruption, while preserving consistency and integrity within the system [1, 2]. Consequently, dynamic reconfiguration can foster the dependability of a system, an attribute that is especially important for critical systems in domains such as air traffic control, energy, disaster management, and health care. Systems in these scenarios are required to maintain a high level of availability, so that stopping them is not an option due to financial costs, physical damages, or even threats to life and safety of people.

Considering dynamicity and dependability while conceiving the architecture of a software system has grown in importance due to the complexity of emerging applications, mainly in critical domains. Nevertheless, these concerns are often handled late in the development process despite software architectures are considered the backbone for any successful system. A system without an adaptable architecture will degenerate sooner than a system based on an architecture that takes changes into account. Furthermore, software architectures are expected to document and allow reasoning about changes that might occur during system execution, as well as to provide basis for its evolution.

Dynamic software architectures are those that describe how architectural elements (components and connectors) can be created, interconnected, and/or removed during system execution. In order to support architecture-based runtime evolution, architecture description languages (ADLs) need to provide specific features to model dynamic changes and techniques for effecting them into the running system [3]. However, a major challenge for ADLs is the ability of describing dynamic software architectures under both structural and behavioral viewpoints, as well as encompassing runtime changes on the structure and behavior of the architecture itself and its components/connectors. Most of the existing ADLs assume that architectures are static and few ADLs support an expressive description of dynamicity concerns.

In another perspective, a recurrent problem of almost all ADLs is the gap between architecture descriptions and respective implementations. As software architectures are typically defined independently from implementation, ADLs often entail architectural mismatches and inconsistencies between architecture and implementation. Therefore, even though a system is initially built to conform to its intended architecture, its implementation may become inconsistent with the original architecture. This problem becomes worse with the emergence of new generation programming languages because most ADLs do not capture the new features of this type of languages, which are intended to take advantage of the modern multicore and networked computer architectures. For these reasons, architectural representation and system implementation need to be continuously synchronized aiming to avoid architectural drift.

978-1-4799-1922-2/15 $31.00 © 2015 IEEE
DOI 10.1109/WICSA.2015.21

The first goal of this paper is to present the dynamic reconfiguration support provided by π-ADL [6], a formal language based on the π-calculus process algebra [7] for specifying software architectures under structural and behavioral viewpoints. This language was designed to support a comprehensive description of dynamic software architectures and their automated rigorous analysis with respect to functional and non-functional properties. In this paper, we introduce architectural-level primitives for specifying *programmed reconfiguration*, i.e., changes described at design time and triggered at runtime by the system itself under a given condition or event [2]. Programmed reconfiguration can be specified in π-ADL by using two different approaches. The first approach is *exogenous*, in which an entity has control over all elements of the software architectures and it is in charge of applying the changes on the whole structure. In turn, the second approach is *endogenous* and represents a decentralized way in which the architectural elements are able to manage the reconfiguration actions [9].

The second goal of this paper is to tackle the existing gap between descriptions of dynamic software architectures and their respective implementations as a continuation of our previous work [10], which had not addressed dynamic reconfiguration issues. In order to support this transition from architecture description to implementation, π-ADL was integrated to the Go programming language [11]. Go was chosen to serve as implementation language because it is an easy general-purpose language designed to address the construction of scalable distributed systems and handle multicore and networked computer architectures, as required by new generation software systems. Furthermore, Go is based on the π-calculus process algebra (the same basis of π-ADL), so that the straightforward relationship between elements of the languages has fostered such an integration.

The remainder of this paper is organized as follows. Section II presents a real-world flood monitoring system used as running example to illustrate our proposal. Section III provides the background of this work. Section IV presents how to describe dynamic software architectures in π-ADL considering both exogenous and endogenous approaches. Section V introduces the mapping of π-ADL architectural descriptions to corresponding implementations in Go and its application to the running example. Section VI discusses related work. Finally, Section VII contains some concluding remarks and give directions to future work.

II. Running Example: A Flood Monitoring System

In rainy seasons, floods are challenging to urban centers traversed by large rivers due to material, human, and economic losses in flooded areas. In order to minimize such problems, a flood monitoring system can support monitoring of urban rivers and create alert messages to notify authorities and citizens in case of an imminent flood. A successful example of such a system is the one deployed to monitor the Monjolinho River in São Carlos, Brazil [12]. This system is based on a wireless sensor network composed of *motes*, tiny hardware/software platforms equipped with embedded CPU, wireless networking capabilities, and simple sensors. Such motes are spread in the proximities of the river and monitor the water level, which is used as an indicator of flooding. In addition, a gateway station analyzes data measured by the motes and triggers alerts when a flood condition is detected. Sensed data are transmitted in a multihop communication, i.e., data sensed by some motes in their sites are successively sent to neighbor sensors, which in turn forward such data to other neighbors until reaching the gateway station. These communications can take place by using wireless network connections, e.g., WiFi, ZigBee, GPRS, etc.

An important requirement for the flood monitoring system regards the highly dynamic environment where it is inserted. Therefore, the flood monitoring system must be able to be dynamically reconfigured to address: (i) efficiency in the use of the available resources, mainly in terms of power consumption and communication; (ii) resiliency of the system in case of unavailability of motes during operation; (iii) precise, accurate measures when detecting floods; and (iv) autonomous, proactive adaptation according to the environmental conditions of the river while minimizing manual intervention. This rich adaptive real-world scenario has motivated us to choose such a flood monitoring system to illustrate our proposal. In this paper, we specify its architecture in π-ADL and show how it can be automatically translated to its respective implementation in Go considering some possible reconfiguration situations.

III. Background

A. The π-ADL architecture description language

π-ADL [6] is a formal language for describing dynamic software architectures under both structural and behavioral viewpoints. It is part of a family of languages designed to formally describe and refine dynamic software architectures, as well as to support automated verification at both design time and runtime. π-ADL is founded upon the higher-order typed π-calculus [7] (hence the name), a computationally complete (Turing-complete) process algebra that provides an universal model of computation. In this perspective, the language extends π-calculus by providing constructs that enable architects to easily express software architectures with computational completeness and high expressiveness.

From the structural viewpoint, an architecture is described in π-ADL in terms of *components*, *connectors*, and their composition to form the system, i.e., the *architecture* itself as a configuration of components and connectors. Components represent the functional elements of the system, whereas connectors manage interactions among components as a component cannot be directly connected to another

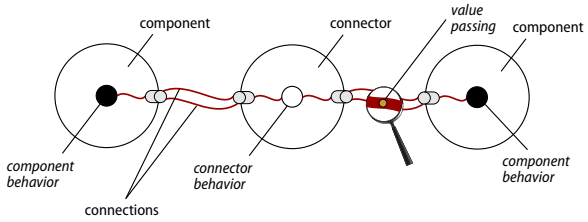

Figure 1. Main architectural concepts of the π-ADL language.

component. From the behavioral viewpoint, both components and connectors comprise a *behavior*, which expresses internal computation and uses *connections* to connect and transmit values. In π-ADL, architectures, components, and connectors are formally specified in terms of *abstractions* over behaviors. Fig. 1 depicts the main architectural concepts of π-ADL. In a black-box perspective, only connections of components and connectors and values passing through them are observable. In a white-box perspective, internal behaviors of such elements are also observable.

As π-ADL is derived from π-calculus, it is grounded on the concept of *communicating processes* [13]. In π-calculus, communications/interactions between concurrent processes take place through *channels*, which are abstractions that enable the synchronization between such processes by sending and receiving messages (values or even channels). In an analogous way, π-ADL provides *connections*, which are abstractions that represent communication channels between architectural elements (components and connectors). By using typed connections, components and connectors can send (output connections) and receive (input connections) any value or even connections. In order to attach a component to a connector, at least a connection of the former must be attached to a connection of the latter. This attachment is called *unification*.

Fig. 2 depicts a simplified architecture of the flood monitoring system and its description in π-ADL[1]. This architecture (`WSNMonitoring`) is specified as a composition of three sensors (`s1`, `s2` and `s3`) and one gateway (`gw`) linked through three ZigBee connectors (`zb1`, `zb2` and `zb3`). In the `Sensor` components, water level data can be measured by using the `sense` input connection or can be received from other sensor via the `pass` input connection, and they are sent via the `measure` output connection. In the `Gateway` component, the `data` input connection is used to receive measured data, which are processed by calling the `triggerAlert` function. If a flood is detected based on the received data, then a message is sent via the `alert` output connection. The architectural elements that compose the architecture are attached via six unifications, each one binding an output connection of an element to an input connection of another.

[1]More details about the syntax of architecture descriptions in π-ADL and its elements can be found in [6].

B. The Go programming language

Go [11] is a new general-purpose language developed at Google to address the construction of new generation software systems, which shall be efficient and deployed on multicore and networked computer architectures. In order to achieve these purposes, the language aims at combining lightweight, ease of use, and expressiveness of interpreted and dynamically typed languages, such as JavaScript and Python, with the efficiency and safety of traditional compiled languages, such as C++ and Java. Moreover, it is possible to compile even a large Go program to native code in few seconds, thus fostering the development of large systems.

One of the main features of Go is the lightweight support for concurrent communication and execution in contrast to the effort required to develop, maintain, and debug concurrent programs in C++, for example. In this perspective, the solution provided by Go is threefold. First, the high-level support for concurrent programming enables programmers to easily develop concurrent systems. Second, concurrent processing is performed through *goroutines*, lightweight processes (similar to threads, but lighter) that can be created and automatically load-balanced across the available processors and cores. Finally, the automatic and efficient garbage collection relieves programmers of the memory management typically required by concurrent programs.

In Go, goroutines communicate by using typed *channels*, which are used as means for sending and receiving values of any type. In a channel communication, sender/receiver channels and their respective goroutines are synchronized at the moment of the communication [13]. Therefore, explicit locking and other low-level details are abstracted away, thus simplifying the development of concurrent programs. Furthermore, due to its theoretical foundations on π-calculus, Go also supports the mobility of channels, i.e., channels are first-class objects that can be transported via other channels.

Due to space restrictions, in Section V we introduce just some elements of Go used in the implementation code generated from architecture descriptions in π-ADL. The interested reader is invited to refer to the complete specification of the language and details about its syntax in [11].

C. Characterizing reconfiguration of software architectures

Wermelinger [9] and Bradbury [14] provide relevant characterizations of dynamic reconfiguration approaches for software architectures. In this section, we briefly discuss the main taxonomic dimensions presented by these authors and that are considered in this work.

Source (or initiation). Dynamic reconfiguration of software architectures can be *programmed* or *ad-hoc*. In programmed (or proactive) reconfigurations, changes are preplanned, foreseen at design time and applied at runtime under a given condition or event. Therefore, the software architect specifies *when* such changes will be realized and *which* operations must be performed. In turn, ad-hoc (or

```
component Sensor is abstraction() {
  connection sense is in(Integer)
  connection pass is in(Integer)
  connection measure is out(Integer)
  behavior is {
    choose {
      via sense receive d : Integer
      via measure send d
    } or {
      via pass receive m : Integer
      via measure send m
    }
  }
}
```

```
connector ZigBee is abstraction() {
  connection input is in(Integer)
  connection output is out(Integer)
  behavior is {
    via input receive m : Integer
    via output send m
  }
}
```

```
component Gateway is abstraction() {
  connection data is in(Integer)
  connection alert is out(String)
  behavior is {
    triggerAlert is function(m : Integer) :
      String {
      unobservable
    }
    via data receive d : Integer
    via alert send triggerAlert(d)
  }
}
```

```
architecture WSNMonitoring
 is abstraction() {
 behavior is {
   compose {
     s1 is Sensor()
     and s2 is Sensor()
     and s3 is Sensor()
     and zb1 is ZigBee()
     and zb2 is ZigBee()
     and zb3 is ZigBee()
     and gw is Gateway()
   }
```

```
   where {
     s1::measure unifies zb1::input
     s2::measure unifies zb2::input
     zb1::output unifies s3::pass
     zb2::output unifies s3::pass
     s3::measure unifies zb3::input
     zb3::output unifies gw::data
   }
 }
}
```

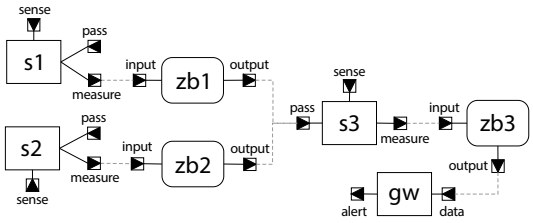

Figure 2. Simplified architecture of the flood monitoring system and its description in π-ADL.

reactive) reconfiguration stands for changes that occur at runtime without being previously planned and are applied through an interface of the system with the external environment in which it is deployed. Both programmed and ad-hoc reconfigurations have their advantages and drawbacks, thus making them complementary approaches that shall be supported by architectural approaches. While programmed reconfigurations can enable a system to be autonomically reconfigured in response to certain conditions, ad-hoc reconfigurations allow applying changes/updates without necessarily foreseeing them in advance. However, it is practically impossible to predict all possible operations that might be required by a system to be reconfigured. On the other hand, ad-hoc reconfigurations must be somewhat constrained, carefully applied in order to avoid architectural erosion. The majority of the works about dynamic software architectures addresses programmed reconfiguration [15].

Operations. In spite of the several nomenclatures adopted in different works in the literature, reconfiguration operations to be applied on the architectural elements of a software system are essentially four [2, 16]: (i) *creation* of architectural element instances; (ii) *removal* of architectural element instances; (iii) attachment of architectural elements; and (iv) detachment of architectural elements.

Management. The management of the reconfiguration process can be either centralized with a special entity or distributed across the architectural elements. We call *exogenous dynamism* the approach characterized by the existence of a central entity (e.g., a configuration manager) that has control over all architectural elements and it is responsible for applying the reconfiguration actions on the architecture. In turn, we call *endogenous dynamism* the approach characterized by the decentralization of the dynamic reconfiguration process in which the architectural elements themselves are able to perform the reconfiguration actions. The main drawback of the exogenous dynamism is the centralization of the reconfiguration process, so that the entity responsible for it may become a bottleneck at the implementation level and reduce the performance of the architecture at runtime. Moreover, reconfiguration actions associated to different architectural elements may be tangled, each one requiring a specific set of operations that must be described and performed independently from each other. In the endogenous approach, the reconfiguration actions may also be tangled with the functionality/behavior of the architectural elements, thus hampering reuse and maintainability. Additionally, the endogenous dynamism requires architectural elements to have knowledge about each other, so that one could regard this as a violation of the basic features of a component or connector. Therefore, it is necessary to establish some sort of trade-off between these approaches for dynamism according to each specific scenario and architecture.

IV. Dynamic Software Architectures in π-ADL

The specification of dynamic software architectures in π-ADL is based on the concept of *active architectures* [8]. Active software architectures are: (i) *dynamic* in that the structure and cardinality of the components and interactions are changeable during execution; (ii) *updatable* in that architectural elements can be dynamically replaced; (iii)

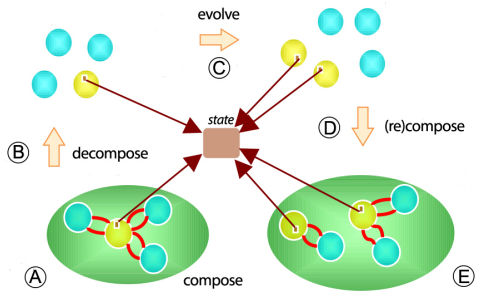

Figure 3. Life cycle for active software architectures [8].

decomposable in that an executing system can be dismantled into its architectural elements; and (iv) *evolvable* in that the specification of components and interactions can be evolved at runtime. Fig. 3 depicts the basic life cycle for an active architecture as envisioned in π-ADL. At the initial stage Ⓐ, the architectural abstractions of the system are *composed* in order to form its software architecture. At stage Ⓑ, the system is *decomposed* to yield its individual architectural abstractions disconnected from each other. Next, such individual architectural abstractions can be *evolved* at stage Ⓒ, e.g., in terms of creating and/or removing elements. Finally, the system architecture is *(re)composed* at stage Ⓓ by composing a new configuration of architectural elements aiming to form the new version of the architecture (stage Ⓔ). It is important to mention that the state of each architectural abstraction (and respective data) is *conserved* at each evolution stage, thus maintaining consistency along the reconfiguration process.

In the light of active architectures, dynamic software architectures can be specified in π-ADL by using two main operators, namely *composition* (`compose`) and *decomposition* (`decompose`). The former is used to compose the architecture by (i) instantiating abstractions corresponding to architectural elements and (ii) unifying connections to attach such elements and allow their synchronization and communication. In turn, the latter is used to dismantle an architecture into a set of behaviors corresponding to the previously composed architectural elements, now detached from each other. Through decomposition, the executing system is broken into its constituent elements, which can be changed and further recomposed to form an evolved system. Although the composition and decomposition operators provided by π-ADL seem to be divergent from the usual reconfiguration actions adopted in most works in literature [2, 16], we argue that they allow specifying dynamic software architectures in a comprehensive way while preserving the formal foundations of the language over the π-calculus process algebra. Therefore, supporting these operators with a suitable underlying formalism fosters formal verification of the specified software architectures, as well as the enforcement of structural, behavioral, and quality properties before, during, and after reconfiguration.

As previously mentioned, our current work is concerned with supporting programmed, foreseen reconfiguration operations under both exogenous and endogenous approaches for describing dynamism in software architectures. In order to exemplify our proposal, consider the flood monitoring system presented in Section II under the two following possible reconfigurations at runtime:

R_1: The battery level of the sensor `s3` is low, thus requiring the replacement of this mote by another one. Although the new mote (`s4`) is near to the other two ones (`s1` and `s2`), it is far from the gateway component, so that it is not possible to use ZigBee for connecting such a mote with the gateway. For this reason, GPRS can be used as it is suitable for communications over long distances.

R_2: A flood was detected based on the data collected by the sensor motes. However, as an evacuation procedure might be expensive, it is necessary to improve the accuracy of the measures aiming at avoiding false positives. For this purpose, unmanned aerial vehicles (UAVs, drones) endowed with digital cameras and WiFi networking capabilities can be used for capturing images from the river to estimate its flow rate. These images can be sent to the gateway, which processes and combines them with data provided by the motes, thus confirming whether a flood is imminent or not.

Fig. 4 shows a partial description of the reconfiguration R_1 in π-ADL with the *exogenous* approach. There are two coexisting architectures, namely the *initial architecture* and the *evolved architecture*, the latter resulted from a reconfiguration applied on the former upon a stimulus. The evolved architecture `WSNMonitoringRec` describes the reconfiguration actions that need to be applied on `WSNMonitoring`, the initial architecture depicted in Fig. 2. The reconfiguration is triggered when a value is sent via the channel `lowb` in the initial architecture, thus indicating that the battery level of the sensor is low. In this case, the coordinating behavior performs an *application* to run `WSNMonitoringRec`, i.e., the evolved version of the initial running architecture (`iarch`) to realize the reconfiguration R_1. First, `WSNMonitoring` is *decomposed* into a sequence of seven detached behaviors (`abs`), each one associated to the architectural elements previously instantiated. Next, the previous instances of the sensors `s1` and `s2`, the gateway `gw`, and the ZigBee connectors `zb1` and `zb2` are composed with a new sensor instance (`s4`) and a new instance of the GPRS connector (`gprs1`). Finally, the connections of these elements are unified.

Fig. 5 shows a partial description of the reconfiguration R_2 in π-ADL with the *endogenous* approach. In this reconfiguration, the gateway component is responsible for applying the changes, i.e., creating one instance of the `UAV` component (`dr`) and one instance of the `WiFi` connector (`wf`) when the flood risk given by the `triggerAlert` function is classified as high or very high. After creating these instances, they are attached by unifying their respective connections. In

```
architecture WSNMonitoringRec is
  abstraction(lowb : connection[Boolean], iarch : Any) {
  behavior is {
    abs is sequence[Behavior]
    abs = decompose iarch      // decomposing WSNMonitoring
    compose {
      s1 is abs[0]             // previous Sensor instance
      and s2 is abs[1]         // previous Sensor instance
      and s4 is Sensor()       // new Sensor instance
      and zb1 is abs[3]        // previous ZigBee instance
      and zb2 is abs[4]        // previous ZigBee instance
      and gprs1 is GPRS()      // new GPRS instance
      and gw is abs[6]         // previous Gateway instance
    } where {
      s1::measure unifies zb1::input
      s2::measure unifies zb2::input
      zb1::output unifies s4::pass
      zb2::output unifies s4::pass
      s4::measure unifies gprs1::input
      gprs1::output unifies gw::data
    }
  }
}

behavior is {                  // controlling behavior
  connection lowb is in(Boolean)
  iarch = WSNMonitoring(lowb)  // initial architecture
  via lowb receive v : Boolean
  if (v == true) then {        // low battery notification
    WSNMonitoringReconf(iarch)
  }
}
```

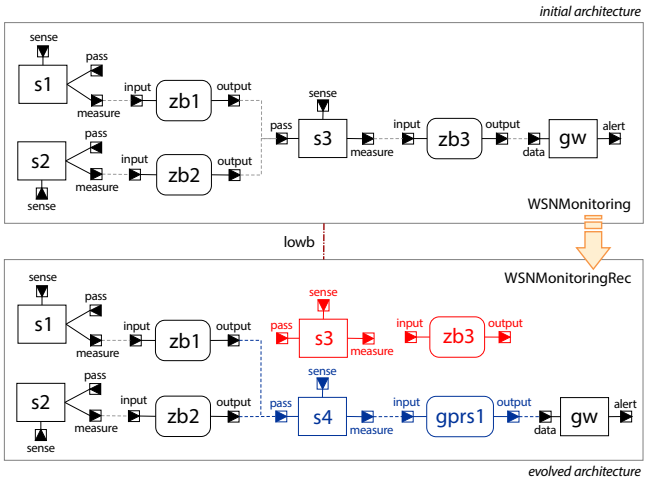

Figure 4. Partial π-ADL description of the exogenous reconfiguration for the flood monitoring system: replacement a mote due to low battery level.

this case, notice that the initial architecture `WSNMonitoring` presented in Fig. 2 does not need to be decomposed as the reconfiguration R_2 does not require detaching architectural elements, but creating and attaching new ones.

V. Mapping from π-ADL to Go

This section presents how to automatically generate source code in Go from π-ADL architecture descriptions. Section V-A defines the correspondences between the elements of π-ADL to Go, whereas Section V-B presents the code generation process. Finally, Section V-C shows the Go source code generated from the architecture description of

```
component UAV is abstraction() {
  unobservable
}

connector WiFi is abstraction() {
  unobservable
}

component Gateway is abstraction() {
  connection data is in(Integer)
  connection image is in(Any)
  behavior is{
    triggerAlert is function(measure : Integer) : String {
      unobservable
    }
    processImage is function(i : Any) : Boolean {
      unobservable
    }
    via data receive d : Integer
    risk is location[String]
    risk = triggerAlert(d)
    if (risk == "High" || risk == "Very high") then {
      compose {
        dr is UAV()      // UAV (drone) component instance
        and wf is WiFi() // WiFi connector instance
      } where {
        dr::output unifies wf::input
        wf::output unifies self::image
      }
      via image receive i : Any
      if (processImage(i) == true) then {
        via alert send "Flood risk confirmed"
      }
    }
    via alert send risk
  }
}
```

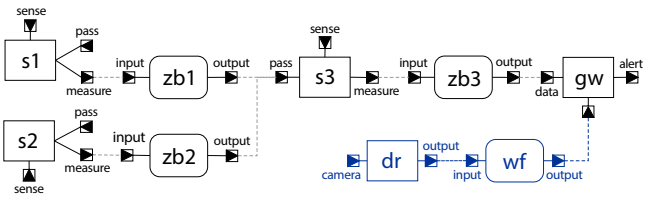

Figure 5. Partial π-ADL description of the endogenous reconfiguration for the flood monitoring system: an UAV is added to increase accuracy and avoid false positives.

the flood monitoring system for the reconfiguration situations presented in Section IV.

A. Correspondences between π-ADL and Go

Table I summarizes the relationships between the main elements of π-ADL and Go, each one described in the following. These correspondences were initially introduced in our previous work [10], which had not addressed dynamism in π-ADL and its transition to Go. In this work, we go a step further by providing implementation support in Go for the reconfiguration operations described in Section IV.

Architectural abstractions and their behavior. In π-ADL, components, connectors, and architectures are created as abstractions over behaviors. In Go, components and connectors are represented as functions called as goroutines (see Section III-B), thus being equivalent to the notion of communicating processes in π-calculus. Such functions

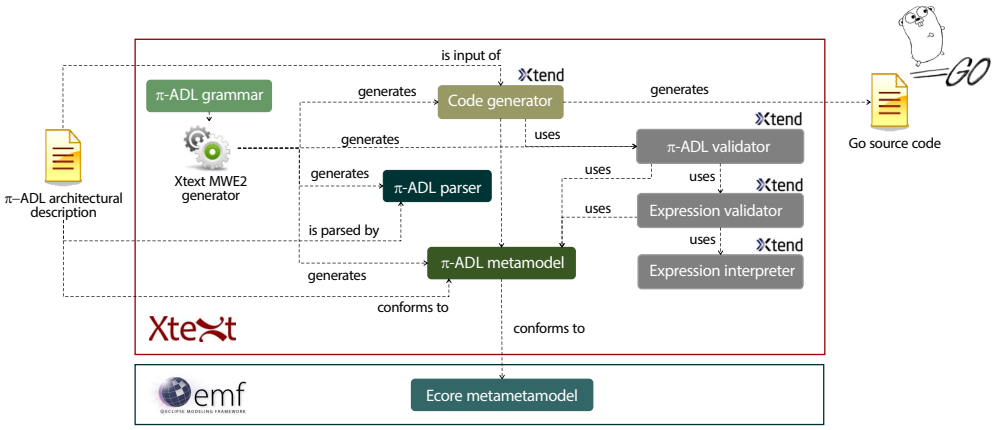

Figure 6. Elements for generating source code in Go from π-ADL architecture descriptions.

TABLE I: SUMMARY OF THE CORRESPONDENCES BETWEEN ELEMENTS OF π-ADL AND GO

π-ADL	**Go**
Architectural abstraction (component, connector, architecture)	Function (goroutine)
Behavior of architectural abstraction	Body of function (goroutine)
Connection	Channel
Instantiation of architectural element	Call to goroutine
Unification of connections	Channels as parameters to goroutines
Detachment of connections	Channels closure
Coordinating behavior	Main function

are declared with the respective names of the architectural abstractions that they represent, and the body of these functions comprises their behavior.

Connections. In π-ADL, connections are used for sending/receiving values between architectural abstractions and their behaviors. Similarly to π-calculus, typed channels in Go are used to send and/or receive values between concurrent processes (goroutines to be synchronized), so that connections in π-ADL are mapped to channels in Go. The type of the values transmitted through a channel is the one specified in the declaration of the connection.

Instantiation of architectural elements. In order to create instances of components and connectors within an architecture, the goroutines that respectively represent these architectural elements are called within the main function. In such calls, the following elements are provided as parameters: the channels representing the connections of the component/connector; and (ii) the output/input channels associated to the connections that will be unified to this component/connector. When performing reconfiguration operations to create a new component/connector, new calls to the respective goroutines that implement them are performed.

Unification of connections. In π-ADL, a connection of a component can be attached to a connection of a connector for enabling such elements to communicate. In Go, this unification process takes place by passing the channels (connections) to be unified as parameters to the goroutines that represent components/connectors, and passing the contents of the sending channel (output connection) to the receiving channel (input connection). The same applies to new unifications performed in a dynamic reconfiguration.

Detachment of architectural elements within decomposition. As described in Section IV, the execution of a decomposition operation automatically removes the existing unifications between the architectural elements previously composed. When mapping from π-ADL to Go, removing these attachments is equivalent to closing the communication channels used to synchronize the goroutines that represent architectural elements. In Go, closing an unbuffered communication channel[2] indicates that no more values will be transmitted through it and leads to an immediate blockage of the goroutines that use such a channel.

Coordinating behavior. In order to enable a system architecture to execute, a coordinating behavior performs an *application* (similar to a call) of the abstraction corresponding to such an architecture. In Go, this behavior represented by the `main` function, which is the first function called when a Go program is executed. Therefore, the main function calls the goroutine that represents the architecture itself, which in turn calls the goroutines associated to the instances of architectural elements (components and connectors).

B. Code generation procedure

Fig. 6 depicts the technical elements related to the generation of source code in the Go from architecture descriptions in π-ADL. We use the Xtext open-source framework [17] to support this process within an environment aimed to assist software architects in the use of the π-ADL language. Xtext covers all aspects of a complete language infrastructure by parsing the textual model written in such a language and allows generating code from it in another language.

[2]In Go, communication is *synchronous* and *unbuffered*: a sending operation does not complete until there is a receiver to accept the value. Therefore, send/receive operations block until the other side is ready [11].

Moreover, this infrastructure is fully integrated with the Eclipse platform, thus taking advantage of useful features such as editor with syntax highlighting, error markers, etc.

From the π-ADL grammar specification in the Extended Backus-Naur Form (EBNF), Xtext automatically generates the π-ADL infrastructure in terms of: (i) a *parser*, which is responsible for the syntactic analysis of the architecture textual description; (ii) the entry point for a *code generator*, which is used to generate code in Go from the architecture description in π-ADL; and (iii) an Eclipse-based *code editor* for assisting the textual description of a software architecture in π-ADL. The code generator makes use of some *validators* to semantically analyze a π-ADL architecture description. In particular, the code generator is implemented by using facilities provided by the Xtend programming language [18], a dialect of the Java programming language that translates a textual model to source code. For this purpose, Xtend uses extension methods and template expressions to specify how a given abstract element in the input model can be translated to its representation in the source code to be generated. Therefore, these mechanisms provided by Xtend were used to translate the abstract elements defined in the π-ADL grammar to their respective implementation in Go based on the correspondences defined in Section V-A. If an architecture description written in π-ADL is correct according to the syntactic and semantic rules of the language, then the respective source code in Go is automatically generated by the π-ADL textual editor.

More details about the mapping process, the Xtext grammar specification of the language, and the Eclipse plug-in for textually describing software architectures in π-ADL and generating source code in Go are publicly available at: http://consiste.dimap.ufrn.br/projects/piadl2go/.

C. Implementing the reconfiguration of the flood monitoring system in Go

Fig. 7 shows an excerpt[3] of the generated Go source code regarding the exogenous reconfiguration specified in Fig. 4. By following the correspondences established in Section V-A, the initial and evolved architectures presented in IV are implemented in Go by the `WSNMonitoring` and `WSNMonitoringRec` goroutines, which are called by the main function (controlling behavior). The decomposition operation performed by the `WSNMonitoringRec` function consists of: (i) receiving the references to the elements previously instantiated in the `WSNMonitoring` function via the `instances` channel; (ii) storing such references in the slice `abs`; and (iii) closing the channels of the elements that will not be used in the evolved architecture – in this case, the sensor `s3` and the ZigBee connector `zb3`, respectively indexed by the positions 2 and 5 in `abs`. Next, the new

[3]Due to space restrictions, we have suppressed part of the generated code aiming at focusing on the reconfiguration operations. Nonetheless, it is available for download at http://consiste.dimap.ufrn.br/projects/piadl2go/.

```
func WSNMonitoring(lowb chan bool,
  instances chan map[string]interface{}) {
  // calls to goroutines representing architectural
  // elements (exported through the instances channel)
}

func WSNMonitoringRec(iarch chan map[string]interface) {
  var abs []map[string]interface{}
  for i := range iarch {
    abs = append(abs, s)
  }

  // closing channels
  <-abs[2]["pass"].(chan int)
  <-abs[2]["measure"].(chan int)
  <-abs[5]["input"].(chan int)
  <-abs[5]["output"].(chan int)

  s4 := map[string]interface{}{
    "sense"   : make(chan int),
    "pass"    : make(chan int),
    "measure" : make(chan int),
  }
  gprs1 := map[string]interface{}{
    "input"  : make(chan int),
    "output" : make(chan int),
  }

  // calls to goroutines (architectural elements)
  go ZigBee(abs[3], abs[0]["measure"], abs[3]["input"])
  go ZigBee(abs[4], abs[1]["measure"], abs[4]["input"])
  go Sensor(s4, abs[3]["output"], s4["pass"])
  go Sensor(s4, abs[4]["output"], s4["pass"])
  go GPRS(gprs1, s4["measure"], gprs1["input"])
  go Gateway(abs[6], gprs1["output"], abs[6]["data"])
}

func main() {
  lowb  := make(chan bool)
  iarch := make(chan map[string]interface{})
  go WSNMonitoring(lowb, iarch)
  v := <-lowb
  if v == true {
    go WSNMonitoringRec(iarch)
  }
}
```

Figure 7. Excerpt of Go source code generated from the π-ADL description regarding the exogenous reconfiguration of the flood monitoring system.

instances of the sensor component (`s4`) and the GPRS connector (`gprs1`) are created. Finally, the unifications take place by passing the channels as parameters to the goroutines (second and third parameters). For instance, the call to the goroutine `GPRS` unifies the connection `output` of the sensor `s4` to the connection `input` of the connector `gprs1`, so that data are sent from the former to the latter.

Fig. 8 shows another excerpt of the generated Go source code regarding the exogenous reconfiguration specified in Fig. 5. As the gateway component is the one responsible for performing the reconfiguration operations, the body of the `Gateway` function implements the creation of the instances `dr` and `wf` with their connections. Next, the call to the `WiFi` goroutine (that represents the WiFi connector) unifies the connection `output` of the UAV `dr` to the connection `input` of the connector `wf`. Similarly, the call to the `Gateway` goroutine unifies the connection `output` of the WiFi connector to the connection `image` of the gateway component.

```
func UAV(conn map[string]interface{},
  fromc, toc interface{}) {
  // ...
}

func WiFi(conn map[string]interface{},
  fromc, toc interface{}) {
  // ...
}

func Gateway(conn map[string]interface{},
  fromc, toc interface{}) {
  // ...
  var triggerAlert func(measure int) string
  triggerAlert = func(measure int) string {
    // ...
  }
  var processImage func(i interface{}) bool
  processImage = func(i interface{}) bool {
    // ...
  }

  d := <-conn["data"].(chan int)
  risk := triggerAlert(d)

  if risk == "High" || risk == "Very high" {
    dr := map[string]interface{}{
      "camera" : make(chan interface{}),
      "output" : make(chan interface{}),
    }
    wf := map[string]interface{}{
      "input"  : make(chan interface{}),
      "output" : make(chan interface{}),
    }

    go WiFi(wf, dr["output"], wf["input"])
    go Gateway(conn, wf["output"], conn["image"])

    i := <-conn["image"].(chan interface{})
    if processImage(i) == true {
      conn["alert"].(chan string) <-
        "Flood risk confirmed"
    }
    conn["alert"].(chan string) <- risk
  }
}
```

Figure 8. Excerpt of Go source code generated from the π-ADL description regarding the endogenous reconfiguration of the flood monitoring system.

VI. Related Work

Languages for describing dynamic software architectures. As reported by Bradbury et al. [15], some ADLs have been proposed in the last years for specifying dynamic software architectures. In particular, most works address programmed dynamic reconfiguration by providing specific reconfiguration primitives at the architectural level to describe when and how the system architecture shall be reconfigured. We briefly present some of these approaches.

One of the earliest ADLs addressing dynamic software architectures is Darwin [4], a declarative language with an operational semantics based on π-calculus. Darwin only allows component instantiation, but not its removal neither the creation/destruction of links.

AADL (Architecture Analysis and Design Language) [5] is a language used to describe both software and hardware architectures of distributed real-time embedded systems. This ADL also allows specifying reconfigurable systems by using state machines that describe *modes* and *mode transitions*: modes represent particular (state) configurations, whereas transitions specify events that enable the system to be reconfigured, i.e., changes from the current mode to another. Therefore, modes and transitions in AADL are programmed, statically defined. However, as AADL is used to specify embedded systems at a lower level, modeling reconfigurations in such a language is constrained to a specific system and platform.

Implementation support for dynamic software architectures. Supporting code generation through the translation of architecture descriptions specified in an ADL to a programming language is not a new research subject. However, it is still a relevant issue mainly due to the concern of maintaining conceptual integrity between architecture representations and corresponding implementation code. Despite the existence of ADLs for describing dynamic software architectures, these languages lack of such an integration between architectural descriptions and implementation. Furthermore, to the best of our knowledge, there is no work on the integration of ADLs targeting dynamic architectures with new generation programming languages. In the following, we briefly discuss some existing work on the integration of architectures with implementation and its limitations.

ArchJava [19] tangles software architecture specifications with Java implementation code. It adds new language constructs for specifying components and connections among them, and their behavior is implemented along with the services that they provide. In terms of dynamicity support, components can be dynamically instantiated similarly to ordinary objects and connected at runtime. However, this approach is limited in that it is more concrete than "pure" ADLs as it has a stronger implementation basis. Moreover, an ArchJava architecture specification cannot be subjected to formal reasoning because the language is essentially based on an informal Java foundation. Furthermore, the generated architectures are to be executed on a single Java Virtual Machine, which is not suitable to the modern multicore and networked computer architectures.

π-ADL.NET [20] is an integration of π-ADL with the Microsoft .NET framework in order to enable the execution of architecture descriptions while preserving formal verification capabilities of the concrete architecture at the implementation level. Despite its intention of bringing a formally founded ADL to an implementation platform, its main limitation that makes it not well suited for new generation software systems regards the lack of counterparts when performing the mappings from π-ADL to the .NET platform. In contrast, this limitation is not observed when mapping from π-ADL to Go due to their common π-calculus foundation, thus fostering straightforward relationships between the elements of these languages. Furthermore, π-ADL.NET also lacks of support for distribution (that is natively supported by Go), thus becoming a constraint when implementing distributed systems. Finally, π-ADL.NET does not support the

decomposition operation used for dynamic reconfiguration, so it is not able to describe dynamic software architectures.

VII. Final Remarks

In this paper, we have addressed the existing gap between architecture descriptions and their respective implementations in the context of large-scale, dynamic software systems. In the architectural perspective, we have introduced the dynamic reconfiguration support provided by π-ADL, a formal ADL for describing dynamic software architectures under structural and behavioral viewpoints. Our approach for addressing dynamic reconfiguration in π-ADL is based on the concept of active architectures: an initial running architecture can be decomposed into its constituent architectural elements, which can be modified and (re)composed to form a new, evolved architecture. These reconfiguration operations are specified and enacted as programmed reconfigurations. Furthermore, π-ADL supports two approaches for managing dynamic reconfiguration, namely: (i) an exogenous approach, in which an entity has control over the architectural elements and centralizes the reconfiguration actions to be applied on the architecture; and (ii) an endogenous, decentralized approach, in which the architectural elements themselves are able to perform the reconfiguration actions.

In another perspective, we have presented the integration of π-ADL with Go, a new general-purpose programming language suitable for constructing large-scale distributed systems deployed in multicore and networked computer architectures, features that are not well-supported by the current mainstream programming languages. Both π-ADL and Go have π-calculus as underlying formalism, thus fostering the straightforward relationship between their elements. As a result, we have defined a comprehensive mapping process from π-ADL to Go aiming at automatically generating source code in Go from dynamic architecture descriptions in π-ADL while avoiding architectural mismatches and inconsistencies mainly as the architecture evolves. In order to illustrate our proposal, we have applied such a translation process to a real-world flood monitoring system.

As future work, we intend to evaluate our approach in other real-world scenarios. We also intend to support ad-hoc reconfigurations, as well as to go towards a runtime support to both types of reconfigurations (programmed and ad-hoc), thus establishing a causal connection between the architectural and runtime levels.

References

[1] J. Kramer and J. Magee, "Dynamic configuration for distributed systems", IEEE Transactions on Software Engineering, vol. 11, no. 4, pp. 424-436, Apr. 1985.

[2] M. Endler and J. Wei, "Programming generic dynamic reconfigurations for distributed applications", Proc. of the 1992 Int. Workshop on Configurable Distributed Systems. USA: IET, 1992, pp. 68-79.

[3] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages", IEEE Trans. on Software Engineering, vol. 26, no. 1, pp. 70-93, Jan. 2000.

[4] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, "Specifying distributed software architectures", in Proc. of the 5th European Software Engineering Conf., W. Schäfer and P. Botella, Eds. LNCS 989. UK: Springer London, 1995, pp. 137-153.

[5] P. H. Feiler and D. P. Gluch, Model-Based Engineering with AADL: An introduction to the SAE Architecture Analysis and Design Language. USA: Addison-Wesley, 2012.

[6] F. Oquendo, "π-ADL: An architecture description language based on the higher-order typed-calculus for specifying dynamic and mobile software architectures", ACM SIGSOFT Software Engineering Notes, vol. 29, no. 3, pp. 1-14, May 2004.

[7] R. Milner, Communicating and mobile systems: The π-calculus. USA: Cambridge University Press, 1999.

[8] R. Morrison, G. Kirby, D. Balasubramaniam, K. Mickan, F. Oquendo, S. Cîmpan, B. Warboys, B. Snowdon, and R. M. Greenwood, "Support for evolving software architectures in the ArchWare ADL", Proc. of the 4th Working IEEE/IFIP Conf. on Software Architecture. USA: IEEE Computer Society, 2004, pp. 69-78.

[9] M. A. Wermelinger, Specification of software architecture reconfiguration, Ph.D. Thesis, Universidade Nova de Lisboa, Portugal, 1999.

[10] E. Cavalcante, F. Oquendo, and T. Batista, "Architecture-based code generation: From π-ADL architecture descriptions to implementations in the Go language", in Proc. of the 8th European Conf. on Software Architecture, P. Avgeriou and U. Zdun, Eds. LNCS 8627. Switzerland: Springer International Publishing, 2014, pp. 130-145.

[11] The Go Programming Language, http://golang.org.

[12] D. Hughes, J. Ueyama, E. Mendiondo, N. Matthys, W. Horré, S. Michiels, C. Huygens, W. Joosen, K. L. Man, and S.-U. Guan, "A middleware platform to support river monitoring using wireless sensor networks", Journal of the Brazilian Computer Society, vol. 17, no. 2, pp. 85-102, Jun. 2011.

[13] C. A. R. Hoare, "Communicating sequential processes", Commun. of the ACM, vol. 21, no. 8, pp. 666-677, Aug. 1978.

[14] J. S. Bradbury, Organizing definitions and formalisms for dynamic software architectures, Tech. Report, Queen's University, Canada, 2004.

[15] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger, "A survey of self-management in dynamic software architecture specifications", Proc. of the First ACM SIGSOFT Workshop on Self-Managed Systems. New York, NY, USA: ACM, 2004, pp. 28-33.

[16] N. Medvidovic, "ADLs and dynamic architecture changes", Joint Proc. of the Second Int. Software Architecture Workshop and the 1996 Int. Workshop on Multiple Perspectives in Software Development. USA: ACM, 1996, pp. 24-27.

[17] Xtext, http://www.eclipse.org/Xtext/.

[18] Xtend, https://www.eclipse.org/xtend/.

[19] J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: Connecting software architecture to implementation", Proc. of the 24th Int. Conf. on Software Engineering. USA: ACM/IEEE Computer Society, 2002, pp. 187-197.

[20] Z. Qayyum, Realization of software architectures using a formal language: Towards languages dedicated to formal development based on π-ADL, Ph.D. Thesis, Université de Bretagne-Sud, France, 2009.