libDGALS: A Library-based Approach to Design Dynamic GALS Systems

Wei-Tsun Sun, Alain Girault, Zoran Salcic, Avinash Malik

libDGALS: A Library-based Approach to Design Dynamic GALS Systems

Wei-Tsun Sun[1,2], Alain Girault[1,2], Zoran Salcic[3], and Avinash Malik[3]

[1]INRIA
[2]Université Grenoble Alpes, F-38000 Grenoble, France
[3]Department of Electrical and Computer Engineering, University of Auckland, New Zealand

***Abstract* – We tackle the problem of designing and programming dynamic and reactive systems with four objectives: being based on a formal model of computation, using different types of concurrency, being efficient, and tolerating failures. The challenge lies in the fact that good formal models with very high level of abstraction generally result in non-efficient implementations. We propose a 'C' based library approach following the formal *Dynamic Globally Asynchronous Locally Synchronous* (DGALS) model of computation. We show how a DGALS system can be dynamically constructed from concurrent behaviors on distributed platforms thanks to the DGALS paradigm. Finally, our experimental results clearly indicate the large execution time and memory footprint gains compared to the current state of the art approaches.**

1. Introduction

In this paper we introduce the libDGALS approach for designing *Dynamic Globally Asynchronous Locally Synchronous* (DGALS) programs using a set of specific API in the C language. The motivation of this work stems from the desire to ease the programming of dynamic and reactive systems (DRS). Most DRSs have some common features: (1) a distributed execution environment, (2) system functions are implemented as concurrent behaviors and (3) concurrent behaviors may have a limited lifetime and can be created and terminated dynamically. A complex DRS is hard to design and implement because of the concurrent executions of various sensors and actuators, synchronization, and communication of concurrent behaviors, as well as the heterogeneous and distributed nature of the execution platforms. To program concurrency within a DRS, using programming languages such as C, C++, or Java, is considered too difficult [1]. In contrast, DRS designs that comply with a formal *Model of Computation* (MoC) allow one to validate and even verify the correctness of critical components of the system. A correctly chosen formal MoC also allows designing a complex system in a modular way by composing simpler parts. For example, behaviors that are executed on distributed sensors and actuators execute at their own pace and hence are more reasonable to implement with *asynchronous* concurrency. In contrast, *synchronous* concurrency is a better choice for composing concurrent behaviors that are running on a *single* computation node to reduce communication overheads over the networks. Synchronous concurrent behaviors communicate with each other more frequently, and at the same time the *synchrony hypothesis* guarantees key system properties, e.g. deterministic behavior [2]. A combination of the two, the GALS [3][4] MoC, lends itself well to a significant number of complex DRSs. To address the dynamic nature of the majority of DRSs, the GALS MoC naturally extends from the static to the dynamic case, called Dynamic GALS or DGALS [5].

The main contributions of this articles are: (1) the presentation of the novel software library to program DGALS systems in C; our library, libDGALS, hides the traditional and explicit thread communications and synchronizations from the designers to ease and make more reliable the programming; (2) the realization of the dynamic behavior creation and the mobility with reactivity on top of the DGALS MoC; (3) the dynamic creation of communication channels between behaviors, which is absent in the other existing DGALS implementations; (4) an efficient implementation with small memory footprint that also utilizes the multi-core architecture on which it can run; and (5) the capability of creating the DGALS systems in a distributed setting.

The rest of the paper is organized as follows: Section II describes the DGALS MoC. Section III presents the libDGALS API to design DGALS systems from the C language to construct dynamic and self-adaptive systems. Details of libDGALS implementation are given in Section IV. Section V presents the results of experiments and comparison of libDGALS with a pure language based approach. Section VI discusses related work and conclusions are given in Section VII.

2. Background

2.1. The intuitive semantics of the DGALS MoC

The aim of the DGALS MoC is to provide a righteous programming model in which programmers can mix synchronous and asynchronous concurrent behaviors, and can manage the dynamicity of the system where behaviors and communication links can be created at run time. The DGALS MoC is built upon the GALS MoC, which it extends with features for designing dynamic and distributed systems. The only known approach to designing such systems is Dynamic SystemJ (*DSystemJ*) [5] which extends SystemJ [6], which is used for systematic design of a complex GALS systems. We address the features lacking in DSystemJ, and use Figure 1 as the illustration of a typical DGALS system, with synchrony, asynchrony and dynamicity, which cannot be implemented in DSystemJ but can be implemented with libDGALS.

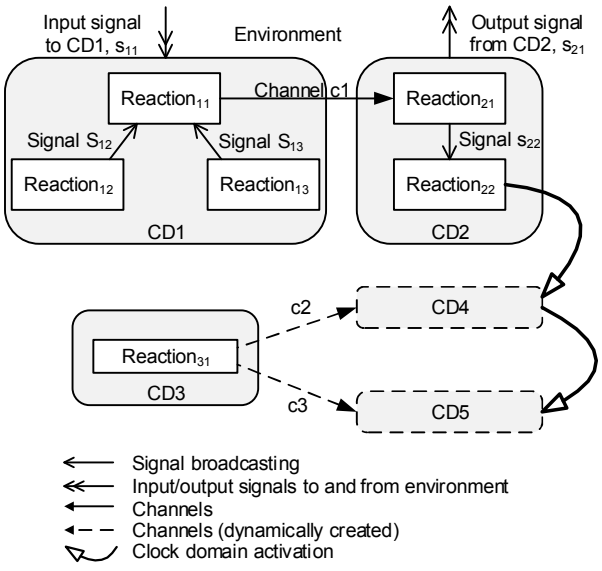


Figure 1. An overview of a DGALS system.

2.1.1. Synchrony in the DGALS MoC

Basic behaviors in DGALS systems are reactive and they interact with the environment continuously, hence they are called *reactions*. A reaction itself is a purely sequential execution unit (a function in C code). Concurrency is achieved by composing reactions into a synchronous product [6]. A group of reactions that execute synchronously comply with the synchronous reactive MoC [7]. The resultant synchronous island is called a *Clock Domain* (*CD*), for instance, CD1 and CD2 in Figure 1. In the synchronous MoC, execution progresses as a sequence of logical ticks, such that all the computations within one tick are viewed as being simultaneous and in *zero* time.

Reactions in a clock domain communicate via *internal signal* broadcast as in Esterel [2]. We use unidirectional to depict a signal emission from the emitter to the target (see Figure 1), possibly branching when there are several targets. Once emitted, a signal is broadcasted to all the other reactions of the same CD in the current tick. Signals interfacing with the environment are sampled and emitted at the beginning and the end of the tick, respectively.

2.1.2. Asynchrony between clock domains

Each CD runs at a different speed and reactions from different CDs communicate via channels. A channel is point to point, unidirectional, and uses CSP rendezvous [7] to guarantee data delivery between reactions.

2.1.3. Dynamic creation of CDs and their communication

The dynamicity of the DGALS systems comes from the creation of CDs at runtime. This is termed *activation* of CDs. To allow communication between reactions of the newly created CD with the existing ones, channels need to be added at runtime. In Figure 1, CDs wrapped in dashed-lines did not exist at system start-up. CD4 is activated by CD2 and communicates with CD3 via the channel c2, also created at run-time. Similarly CD4 activates CD5 which communicates with CD3 through dynamically created channel c3. The system shown in Figure 1 could be programmed with DSystemJ provided that the two channels c2 and c3 be created *statically* so that they be used by CD4 and CD5 (clock domains that are created *dynamically*) to receive messages from CD3. However, in many cases it is not possible to know in advance how many clock domains will be created dynamically, hence channels created statically may not suffice. To overcome this limitation, libDGALS offers the possibility to create channels dynamically.

2.2. Language-based and library-based approaches

There are two major approaches to ease system design; one is the language-based approach and one is the library-based approach. Compilers are essential to generate implementation of the design in the language based approach.

Libraries are extensions of existing languages. Libraries take advantage of the language so that it is not necessary to design a new compiler. Run-time support provided by libraries has more flexibility than static checking in a language-based approach. It is also possible to bind with other libraries to merge different design concepts. A library can be underlined by a specific MoC or multiple MoCs. Even though libraries do not force designers to construct a correct program as a compiler does, they still offer extra features and programming constructs that help programmers to avoid errors.

Unlike the language-based approach of DSystemJ, we take the library-based approach in libDGALS. This is because traditional programming languages (such as C) are fairly popular among main-stream programmers. The other motivation is that the development cycle (iterations of prototyping) of the library is shorter than a language-based approach. Adding and integrating an extra module to the library is relatively straightforward. It does not mean the language- and the library-based approaches are mutually exclusive. It is

Table 1. The programming interface of libDGALS

Constructs	Descriptions
Scope definition	
CDPlugin { … } END_CDPlugin	Defines the scope of a CD plug-in
REACTION_FUNCTION (rName*, output(s)*) { … } END_REACTION_FUNCTION	Defines the scope of a reaction named rName. If the reaction emits signal *s,* then *s* has to be registered with the *output* keyword.
Entity creation (used in the scope of CDPlugin)	
createReaction(fName, active, rName, *setArgument(aName1,* *aValue1 ….)*);	Creates a reaction named *rName*, which is defined as function *fName*. The activation is set by *active*. Optional arguments are passed through the *setArgument* keyword, given as name-value pairs.
createSignal(sName);	Creates a signal named *sName*.
createInputSignal(sName, sHandler, *sHandlerArguemts.*); createOutputSignal(sName, sHandler, *sHandlerArguments* …);	Creates an input (resp. output) signal named *sName*, to receive from (resp. to output to) the environment. *sHander* is the handler function which takes *sHandlerArguments* for configuration.
createChannel(cName);	Creates a channel *cName*.
Statements for synchrony (used in reaction function)	
getArgument(aName);	Gets the argument *aName*.
pause( );	Finishes the current instance
emit(sName, sValue);	Emits signal *sName* with value.
await[Immediate](sName);	Awaits for signal *sName*.
[weak|strong]Abort(sName, abortID) { … } endAbort(sName, abortID);	Creates a scope which can be preempted when signal *sName* is present. *abortID* is used to identify the scope.
setTrap(tName, trapID) { … exitTrap(tName); … } endTrap(tName, trapID);	Similar to abort, but allows preemption to be triggered within the scope.
present(sName) { … }	Checks the presence of the signal *sName*, if present, the scoped code will be executed.
value(sName); preValue(sName);	Obtains the current (resp. previous) signal value.
Statements for asynchrony (used in reaction function)	
send(cName, data, serializer); receive(cName, data, deserializer);	Sends through channel *cName*. The data will be serialized with the *serializer*. This also applies to receive call.
Statement for dynamicity (used in reaction function)	
initCDArgument(cdArg); addCDArgument(cdArg, argName, argValue, argSerializer);	Initializes and adds arguments *cdArg* for CD activation. Each argument will be serialized through *argSerializer.*
activateCD(pTarget, plName, cdName *chName, cnName,* *setCDArgument(cdArg)*);	Activates CD *cdName* from the plug-in *plName* on program *pTaget*.
terminateCD(cdName);	Terminates CD *cdName*.
addChannel(cName);	Creates a channel dynamically.
Statement for dynamicity (used in CDPlugin)	
getCDArgument(argName, data, argDeserializer);	Obtains the CD argument.

usually convenient to quickly prototype designs and ideas using a library based approach, which can later be formulated in a domain specific language like DSystemJ.

3. Programming DGALS systems with libDGALS

In this section, the main contributions of libDGALS are detailed: (1) the intuitive programming interface (as shown in Table 1) to construct DGALS systems, (2) dynamic creation of CDs for scalability, (3) the dynamic channel creation which is unique in libDGALS for existing and newly created CDs, and (4) the capability of implementing strong-mobility for self-adaptive.

According to the DGALS MoC, two entity levels exist: clock domain (CD) and reactions. A CD consists of several reactions, and can spawn further CDs dynamically. To these two levels, libDGALS adds the notion of *DGALS system*, which is the set of all active CDs, and the notion of *DGALS program*, which is a subset of CDs deployed over a given computing location. Each DGALS program is configured with an XML file. DGALS programs can run on one computing location or on multiple physical machines in a distributed (networked) system. Therefore the DGALS system in Figure 1 can be implemented as two DGALS programs executing on two different processors as shown in Figure 2.

The creation of CDs is supported through the uses of *CD plug-ins*. A CD plug-in encapsulates all the information of a CD such as its reactions, channels, and signals, which are instantiated upon activation of a CD, which we call a *CD plug-in instance*. A CD plug-in can be used to instantiate one or more CDs. CDs instantiated from the same CD plug-in can be then customized by receiving extra information from channels. The construction of CDs and configuration of DGALS programs is detailed in this section.

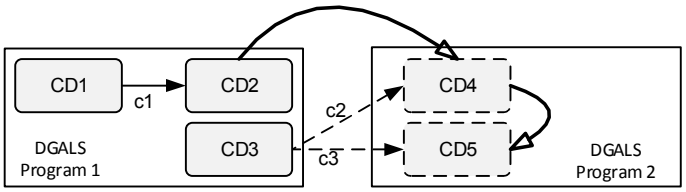


Figure 2. The distributed DGALS system from Figure 1.

3.1. Building clock domain plug-ins

Listings 1 and 2 illustrate how to build CD plug-ins encapsulating CD1 and CD2 from Figure 1. A CD plug-in consists of two parts: the definition of reaction behaviors (reaction functions) and the definition of the CD plug-in. Generally, we follow a bottom-up design approach: defining the reaction functions first, and then assembling them in the plug-in definition later.

In order to use the constructs provided by libDGALS, the mandatory header file libDGALS.h is included first (line 1). A definition of a reaction starts by defining its name using macro *REACTION_FUNCTION* (lines 2, 9, and 14) and ends with macro *END_REACTION_FUNCTION* (lines 8, 13, 16). If a reaction emits a signal, the signal must be declared with the *output* construct as shown in lines 9 and 14 for signals s12 and s13, respectively.

The behavior of a reaction can be described using usual C statements and a set of *reactive statements* (which are colored red in the listings). Examples of reactive statements include *await* (waiting on the presence of a signal, line 4), *parallel* (fork out and then wait for the child-reactions to finish, line 5), *value* (read the value of a signal, line 6), *pause* (explicit end of tick, line 11), and *emit* (for broadcasting the presence of a signal, line 12).

The above mentioned reactive constructs are synchronous and effective within the CD. To access the channel communication between CDs, one must use *send* statement (line 7) to transfer the message (in this case an integer) to the receiver through the specified channel (c1). The arguments to the send call are: (1) the name of the channel, (2) the data to transfer, and (3) the type of the data (can be primitive or struct-typed)

A CD plug-in is defined between *CDPlugin* (line 17) and *END_CDPlugin* (line 24). Channels are created through the use of *createChannel* (line 18) which takes the name of the channel as the argument. To create an input signal that samples the environment, *createInputSignal* is used. The arguments given are the name of the signal, and the type of the signal. Internal signals are created via *createSignal* (lines 20). Reactions are then instantiated with *createReaction* (lines 21-23) with arguments as follows: (1) the reaction function used to create the reaction, (2) status of the created reaction – 1 means active, 0 means dormant and will be activated by its parent reaction with *parallel*, and (3) the name of the reaction.

Listing 1. The Clock Domain 1.

```
1   #include "libDGALS.h"
2   REACTION_FUNCTION(RFunc11) {
3     while(1) {
4       await("s11"); // await for signal s11
5       parallel("Reaction12", "Reaction13");
6       int valueToSend = (int)value("s13");
7       send("c1",valueToSend, int);
8   }} END_REACTION_FUNCTION(RFunc11)
9   REACTION_FUNCTION(RFunc12, output("s12")) {
10    int result = someComputation();
11    pause(); // wait for the next tick
12    emit("s12", result); // emit the signal s12
13  } END_REACTION_FUNCTION(RFunc12)
14  REACTION_FUNCTION(RFunc13, output("s13")) {
15    ..... // other computations
16  } END_REACTION_FUNCTION(RFunc13)
17  CDPlugin(CD1) {
18    createChannel("c1");
19    createInputSignal("s11", TCPInput,   IP  ,   PORT  );
20    createSignal("s12"); createSignal("s13");
21    createReaction(RFunc13, 1, "Reaction11");
22    createReaction(RFunc12, 0, "Reaction12");
23    createReaction(RFunc11, 0, "Reaction13");
24  } END_CDPlugin(CD1)
```

The same design process is followed to describe CD2, as shown in Listing 2. For receiving message from channel c1 at the reaction function RFunc21, a place-holder of the receiving message needs to be declared (line 3) prior to the *receive* call (line 4). The arguments of *receive* are the same as *send*.

As shown in Figure 1, CD2 activates CD4. To create a CD at run-time, *activateCD* (line 13) is called with the following arguments: (1) the destination DGALS program, (2) the name of the CD plug-in, (3) the mapping name of the activated CD, and (4) the mapping names of the channels if there are any. In this case, the CD4 is activated on the DGALS Program 2 with the same named CD plug-in. The name of the channel is mapped from "ch" to "c2" which will be used between CD3 and CD4. The destination DGALS program is specified in the configuration file, which will be detailed in the next section.

Listing 2. The code snippet of CD2

```
1   REACTION_FUNCTION(RFunc21, output("s22")) {
2     while(1) {
3       int receivedValue = 0;
4       receive("c1", receivedValue, int);
5       if(receivedValue >   SOME VALUE  )
6         emit("s22");
7       ..... // other computations
8   } END_REACTION_FUNCTION(RFunc21)
9   REACTION_FUNCTION(RFunc22, output("s21")) {
10    while(1) {
11      pause();
12      present("s22") {
13        activateCD("DGALS_Prog2", // target
14                   "CD4", "CD4",  // map CD name
15                   "ch", "c2");   // map channel name
16        emit("s21", 1);
17  }}} REACTION_FUNCTION(RFunc22)
```

libDGALS enables such dynamic channel creation via the *addChannel* statement (line 2, Listing 3). The resulting channel name has a global scope. Any existing or newly created CD can use it to communicate with other CDs.

Listing 3. The code snippet of CD3 adds channel at runtime.

```
1  REACTION_FUNCTION(ReactionInCD3) {
2    addChannel("c2"); addChannel("c3");
3      while(1) {
4      int kickStart = 1;
5      send("c2", kickStart, int);
6      send("c3", kickStart, int);
7  }} END_REACTION_FUNCTION(ReactionInCD3)
```

3.2. Deploying the CD plug-ins on distributed DGALS programs

Each DGALS program is accompanied by its configuration written in the XML format. The configuration consists of (1) the port used by the DGALS program, (2) CDs which are activated when the DGALS program starts, and (3) the list of known DGALS programs. Listing 4 shows the configuration file for the DGALS program 1 of Figure 2. Each key-value pair represents settings for the specified compartment, or a scope of a component. The <port> (line 1) indicates the port number that Listener of this DGALS program will listen on. The configuration of a statically loaded CD starts with <CDPlugin> (lines 2-5), along with the optional (if overwritten) mapping names of the plug-in. As said above, it is possible that a DGALS system is distributed over a number of DGALS program.

A list of the DGALS programs is provided in the configuration (e.g., lines 6-9) with their corresponding IP and PORT. This is used for locating CD and channels, as well as presenting the destination DGALS program of the *activateCD* call. This separates the construction of the overall DGALS system and the actual distribution of the systems. For example, both DGALS programs in this example can be assigned to the same node (hence the same IP and the port).

Listing 4. The XML configuration of the DGALS program.

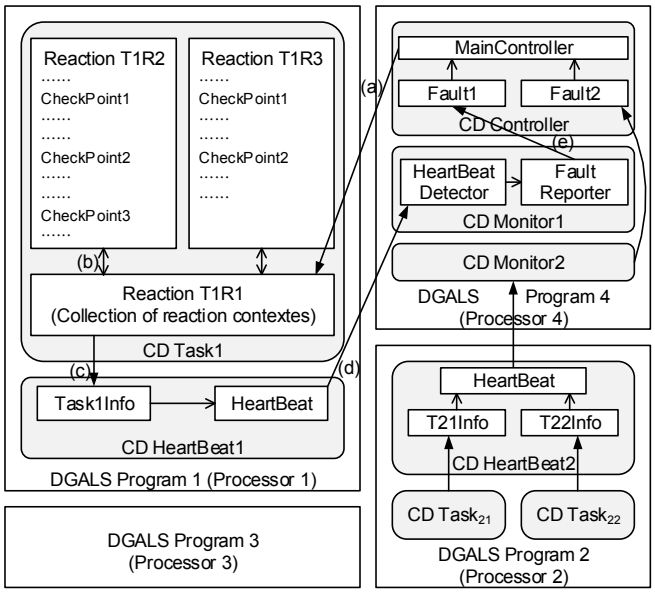
```
1  <port>port_for_the_listener</port>
2  <CDPlugin>CD1
3    <rename>CD1</rename></CDPlugin>
4  <CDPlugin>CD2</CDPlugin>
5  <CDPlugin>CD3</CDPlugin>
6  <DGALSProgram>  IP AND PORT 1
7    <alias>DGALS_Prog1</alias></DGALSProgram>
8  <DGALSProgram>  IP AND PORT 2
9    <alias>DGALS_Prog2</alias></DGALSProgram>
```

3.3. Designing a fault-tolerant system with libDGALS

Strong mobility is the ability for a CD to migrate from a program to another one and resume execution at the point where it stopped. We use an example of a self-adaptive system to show the implementation of strong mobility, execution is resumed after migration, of CDs. Figure 4 shows a fault tolerant system, that is, in case the processor which executes the task(s) fails, the affected task (which is implemented as CDs) will be resumed on another processor.

As shown in Figure 3, there are four available processors, such that each of them executes a DGALS program initially. The behaviors of the system are governed by CD Controller, which collects information from the processors and responds when required. Each of the three main tasks is implemented as a CD, namely Task1 (on processor 1), Task21, and Task22 (on processor 2), respectively. Each CD is further refined into a number of reactions, for example, CD Task1 consists of three reactions T1R1, T1R2, and T1R3.

Each reaction executes a number of checkpoints, such that it can resume its execution following a migration by rolling back to the last saved checkpoint. At each check point, the context, i.e. the instruction counter (the address of the program) and a set of local variables, are stored. This information is collected by a dedicated reaction in the CD, for example T1R1 in CD Task1. This information is sent to the heart-beat CD, which sends messages regularly (at a pre-defined time interval) to monitoring CD indicating that the processor is still functional. The message sent to the monitor CD consists of the timestamp of the message as well as the context of the CD(s). For example, CD HeartBeat2 collects information from CD Task21 and Task22 on processor 2, and notifies the corresponding CD Monitor2 on the processor 4. If a monitoring CD has not received the heart-beat in the required time, the processor is considered faulty, and is reported to CD Controller. CD Controller collects the information from the monitoring CDs through two dedicate reactions, Fault1 and Fault2, and makes a decision synchronously and deterministically in the reaction MainController to react to the fault. We use code snippets of CD Task1 and CD Controller, shown in Listing 5, and Listing 6 to show how this system can be constructed in libDGALS.



(a) Provide the initialization information to the task
(b) Collect the context of task
(c) Deliver the collected task contexts
(d) Send the heart beat of the processor
(e) Report the failure of the processor

Figure 3. A Self-adaptive system.

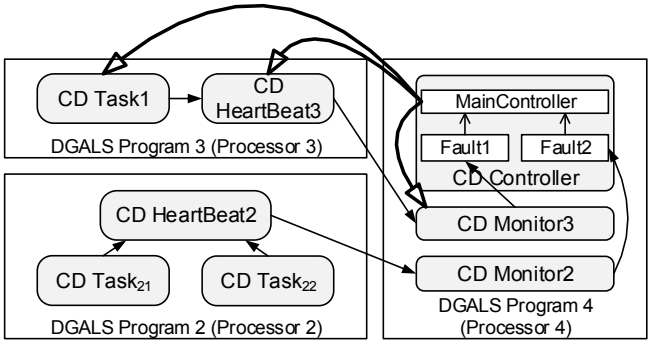


Figure 4. Activate CD Task1 on processor 3 when fault happens.

The reaction MainController is in charge of starting the tasks on faults. At system startup the CD Task1 and Task2 are initialized on their respective processors. During normal execution the context is saved and sent to the MainController via Monitor CDs (Listing 5) at pre-determined check points. Upon fault, the monitoring CDs automatically terminate and pass on the control to the MainController,

which restarts the Tasks on different processor, in the process restoring the saved CD context (Listing 6).

Listing 5. Code snippets of CD Task1.

```
REACTION_FUNCTION(T1R1, output(......)) {
  // receive CDContextInfo when the CD is created
  CDContextInfo *cdci;
  receive("cCDCI1", cdci, CDContextInfo);
  // extract individual context from the CD context
  ContextInfo ciT1R1 = extractContext(cdci, 1);
  ContextInfo ciT1R2 = extractContext(cdci, 2);
  emit("sRCP1", ciT1R2); emit("sRCP2", ciT1R3);
  while(1) {
    // obtain check points from reactions
    awaitImmediate("sCP1"); awaitImmediate("sCP2");
    // construct CD context from reactions, send to HB
    cdci = mergeContext(value("sCP1"), value("sCP2"));
    send("cTask1 to HB", cdci, CDContextInfo);
}} END_REACTION_FUNCTION(T1R1)
```

Listing 6. Code snippets of CD Controller

```
REACTION_FUNCTION(MainController, output(......)) {
  // start tasks with the default context (beginning)
  CDContextInfo *defaultCI;
  send("cCDCI1", defaultCI, CDContextInfo);
  ......
  HBInfo *recoverHBI1 = 0, *recoverHBI2 = 0;
  while(1) { // receive the heart-beat from tasks
    recoverHBI1 = value("sFault1");
    recoverHBI2 = value("sFault2");
    ...... // controller logic based on the fault(s)
    if(activate1) {
      activateCD("Prog3", "Task1", "Task1", ...);
      activateCD("Prog4", "Monitor1", "Monitor3", ...);
      ......
  }
} END_REACTION_FUNCTION(MainController)
```

4. libDGALS implementation

libDGALS can be considered as an extension and an enhancement of [8] which was for designing static GALS systems. Static GALS systems can only exploit multi-core processors, while libDGALS allows both multi-core and distributed memory architectures to be exploited.

Figure 5 and Figure 6 present high level views of the programming stack and the runtime system provided by libDGALS. As illustrated in Figure 5, libDGALS is implemented in C, with the support of threading libraries (can be kernel or user threading), to program CD plug-ins. Other software libraries can be used to describe specialized computations (e.g. image processing) in CD plug-ins. libDGALS has been ported to different thread libraries such as pthread (preemptive) and GNU pth (user-space thread, which does not allow to take advantage from multi-core platform as pthread). In this paper, we are using the pthread version of libDGALS to perform evaluation and analysis.

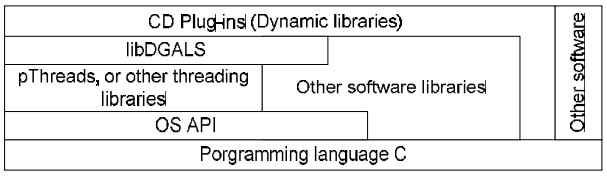


Figure 5. Programming stack of the framework.

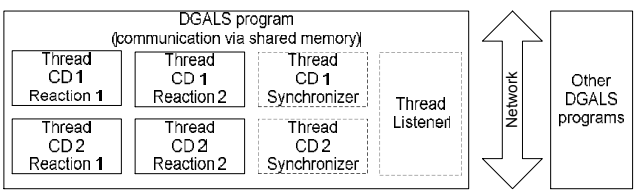


Figure 6. Runtime system of the DGAL programs.

A DGALS program, the runtime system of the libDGALS, is shown in Figure 6. In libDGALS, each CD is implemented as a set of data structures used by reactions. Then, each reaction is implemented as a thread whose execution body is defined by the reaction function described in Section 3.

DGALS programs can communicate with each other over the network. Each DGALS program relies on two kinds of programmer-invisible threads, *Synchronizer* and *Listener*, to make sure that the threads conform to the DGALS MoC.

A *Synchronizer* is a special thread that manages reactions within a CD. It performs the following operations: (1) dynamic resolution of signal dependencies, (2) synchronization of reactions at the CD tick boundaries, (3) maintenance of internal data structures, (4) implementation of the input and output functions used for communication with the environment, and (5) update of the channel statuses to implement rendezvous between reactions belonging to different CDs.

The *Listener* carries out message exchanges with the Listeners of the other DGALS programs to provide: (1) channel communication and (2) dynamic creation of CDs. Shared-memory and TCP based communications are used by the Listener depending upon the configurations of the DGALS programs. Dynamic creations of CDs operate in a similar manner.

5. Experimental results

5.1. Comparisons with existing approaches

We carried out a number of experiments with different examples and physical execution architecture setups to assess the effectiveness of the DGALS approach. Because DSystemJ is the only other language which implements the DGALS MoC, we compare libDGALS with DSystemJ in the metric of code size and runtime performance. In all applications of Figure 7, libDGALS achieves comparable source code size to DSystemJ, which means that libDGALS as concise as a language-based approach. In terms of CD code size, libDGALS models are smaller than the DSystemJ counterparts in all cases. The run-time environment of libDGALS amounts to 140 KB, which is much less compared to DSystemJ, which requires the Java virtual machine.

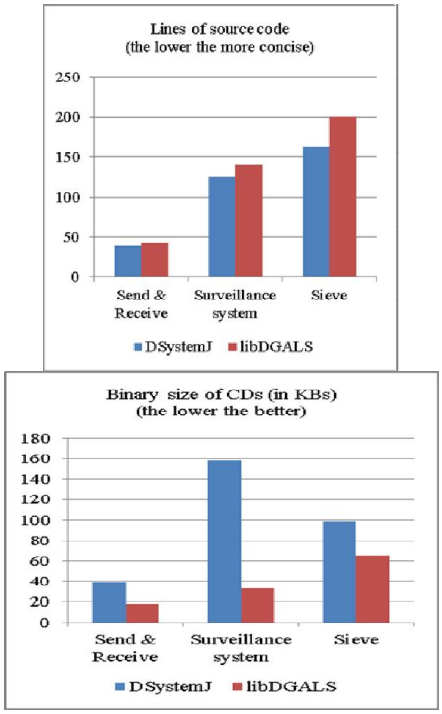


Figure 7. Comparisons between libDGALS over DSystemJ.

Figure 8 shows the execution times between DSystemJ and libDGALS. libDGALS models outperform DSystemJ counterparts. For a simple system (Send & Receive), the DGALS programs are up to 62 times faster compared to DSystemJ, while on a more complex example (Sieve), libDGALS can still achieve 28 times of speedups.

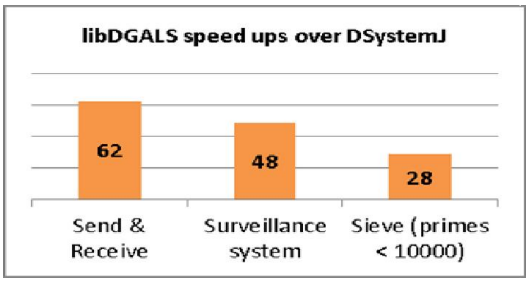


Figure 8. Speedups of libDGALS over DSystemJ: Average ratio of $\frac{\text{execution time of DSystemJ program}}{\text{execution time of libDGALS program}}$ over 1000 runs.

5.2. Multi-core utilization: the N-Body simulation

To explore the modeling features (synchrony, asynchrony, and especially dynamicity) of libDGALS, and to demonstrate that libDGALS can benefit from the multi-core execution platform for better performance, we designed an N-Body simulation (similar to a solar system) as shown in Figure 9. The system consists of a sun, multiple planets orbiting around the sun, and moons orbiting around each planet. The video of the simulation is available online [9].

The Sun and each planet are implemented as a CD. To compute the position of a planet in each iteration, the position, acceleration and velocity of neighboring planets is required. This communication is performed via channels. The number of planets can be increased by clicking the display window dynamically through run-time creation of CDs. The channels are also added dynamically for newly created CDs to enable communication with the existing ones.

Each planet is surrounded by 9 moons. The moons are implemented as synchronous reactions inside each planet CD. The required information to compute the positions of the moons (as well as initial speeds and positions) is broadcast via signals.

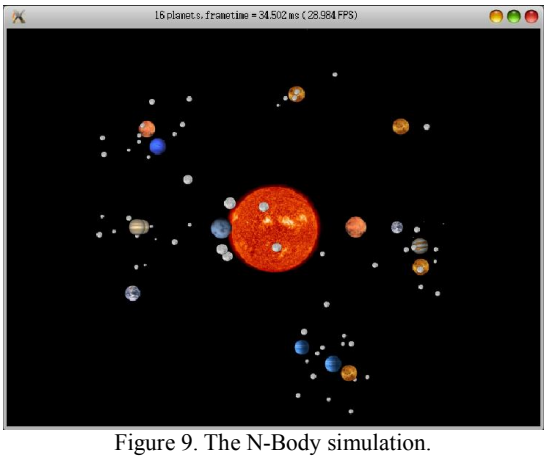


Figure 9. The N-Body simulation.

We experimented with this N-Body simulation on different processor configurations (up to 8 cores) and observed the performance by adding more planets and moons the results are shown in Figure 10. The performance gain, i.e. higher frame rate, can be observed with increasing the number of the cores. The speedup provided by using 8 cores versus 1 core increases when the computing load increases: 3.8 for 10 planets, then 4.4 for 20 planets, and finally 5.6 for 30 planets. We are still investigating the drop of performance when 2 cores are used. We suspect that the overhead of mapping reactions to kernel threads (the implementation of the pthread) introduces overheads, which are more than performance gain in this case.

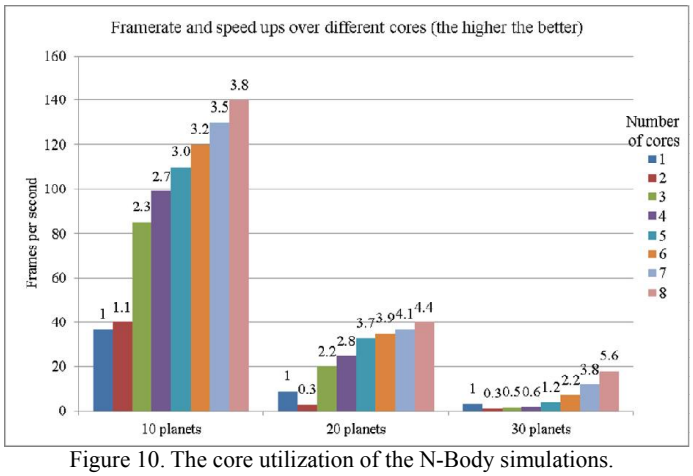


Figure 10. The core utilization of the N-Body simulations.

This N-Body simulation cannot be modeled in DSystemJ because it dynamically creates channels. The N-Body simulation fully captures the features of the DGALS MoC. Within one planet CD, the reactions of the planet and the moons synchronize with each other, but not with the moons of the other planets. All planet CDs communicate with the Sun CD. Finally, because CDs can be activated on different DGALS programs, therefore planets can be distributed over different machines when requiring more computational power.

5.3. Lessons learnt

libDGALS introduces dynamicity by allowing run-time creation of CDs and channels. Our implementation forks a thread for each reaction in a CD. The CD itself is an encapsulation for these reactions. This is in complete contrast with the SystemJ and DSystemJ compilation approaches, which compile away all the synchronous concurrency within a CD into a single execution thread. For libDGALS, because each reaction is mapped to a thread, larger number of reactions within a CD will result in a larger number of threads. In this case, the underlying OS is in charge of scheduling the threads efficiently and achieving better utilization of processor resources. The disadvantage of taking this approach is the use of semaphores (in the libDGALS core, but not in the reaction functions) to ensure read-write communication barriers (SystemJ and DSystemJ resolve dependencies using topological sorting), but as we show in Section 5, using semaphores is a small price to pay compared to the increased program throughput. Using special threads to implement Synchronizers and Listeners for communication with the environment and between different CDs allows us to keep all the impurity (change of state) separate from the logic of the actual program itself. Having pure logic makes reasoning about the program easier, and debugging becomes simpler, too. Using user-space cooperative threading libraries such as GNU pth has less impact on synchronization overheads within a CD but loses the benefit from the multi-core architecture. Mixed thread-mapping strategy should be investigated to improve further the run-time efficiency.

Implementing CD mobility in a ‘C’ based library (or language) is a real challenge, because we are closer to the underlying platform and hence, need to consider a heterogeneous execution platform with varying native architectures (32-bit, 64-bit, Endianness, etc). Our framework is able to accommodate such heterogeneity, by using the standard dynamic library loading approach. We only allow mobility at the CD level because: (1) according to the synchronous MoC it makes

no sense for reactions within a CD to move to another CD and (2) our message passing communication framework allows communication between heterogeneous processes since the communication protocol is standardized using CSP.

Last but not least, because DSystemJ requires the channels to be explicitly declared, it is impossible to have channels created dynamically at runtime. This makes the implementation of the N-Body simulation infeasible in DSystemJ. Creating the channels statically is always possible of course, but this limits the scalability of the design, which also illustrates why the features of dynamicity are important. We believe it would take much longer to implement such features in DSystemJ because of the complexity of the language-based approach.

6. Related works and next steps

Programming frameworks (libraries and languages) targeted at the design of complex systems need to meet a number of requirements to be effectively used by system designers. Safe mechanisms for synchronization and communication between concurrent entities should be primitives in the framework. Thus, a formal MoC, high level abstractions for data fusion and abstract means of accommodating legacy code are also essential.

Languages modeling the GALS MoC (Globally Asynchronous Locally Synchronous) have been proposed [10], but these do not target the same class of dynamic, resource and time constrained systems as us. Dynamic SystemJ (*DSystemJ*) [5] extends SystemJ [6] to a domain of dynamic systems. DSystemJ has a number of limitations: (1) it requires the Java Virtual Machine (JVM), which is significantly abstracted from the underlying platform to efficiently utilize heterogeneous execution architectures, with a heavy price on requirements on memory footprint; and (2) it is unable to create channels dynamically for newly spawned clock domains.

Combining asynchronous concurrent behaviors into bigger systems is also used in the world of *actors* [11], where asynchronous actors communicate with each other using message passing. There are a number of implementations in the form of libraries or frameworks added to existing programming languages [12][13] or included into new concurrent languages [14]. However, they lack the ability to abstractly program reactions on events from the environment, which is essential for reactive systems. *Multi-agent systems*, such as JADE [15] provide reactivity, but at the expense of a significant execution overhead shown in [5].

Many attempts with tools and frameworks based on completely new languages such as X10 [16] and Occam [17] created a gap from the traditional programming world. These approaches do not allow easy interfacing with legacy code and lack many desirable properties, similarly to the actor based approaches. For example, X10 is *not* based on a formal MoC. Occam does not support mobility. Occam also lacks support for complex data-driven computations. Finally, none of these languages provide abstractions, which allow the efficient design of reactive systems, which DRSs are.

Some languages, such as Axum [18], take into account current languages and legacy code, but also rely on powerful and heavy virtual machines (the .NET), which abstract away the underlying platform. Other approaches such as MPI [19] and OpenMP [20] use C/C++, like the proposed DGALS framework, but are either limited to static systems (MPI-1 and OpenMP), or to dynamic systems (MPI-2), without support for mobility and reactivity. Also, they all lack an all-encompassing formal MoC, leaving many issues such as race-conditions to the programmers.

The proposed libDGALS is the first software library-based approach to ease the programming of complex dynamic systems based on a formal MoC. Concurrency model is based on a formal MoC (GALS), so that designers do not have to deal with threads such as pthreads or Java threads. We believe that a C based library also makes integrating legacy code easier. Being based on C makes the library suitable for embedded applications with a small memory an execution time footprint.

One of the possible future works is to investigate the run-time verification to enforce the MoC in a more concrete manner. Currently we are also performing comparisons of different scheduling policies (such as user-space scheduling) to seek further performance improvements.

Because it is possible for a CD to send a name of the channel to the other CD, followed by the fact that a channel can be created dynamically with the received name, the behavior is very similar to the pi-calculus [21]. We are planning to investigate this and write the formal semantics of libDGALS in the near future.

7. Conclusions

We set ourselves the tough challenge of designing a framework, which allows us to build real-world dynamic real-time systems, supporting various forms of concurrency, communication with the environment, and fault tolerance, and most importantly dynamics of concurrent behaviors. Our proposal, libDGALS, is based on the formal DGALS MoC. It allows the composition of synchronous and asynchronous concurrent behaviors with hierarchical relationship. Abstract means of communication are provided using the concepts of reactions to signals from the environment. The libDGALS framework, being based on C, leads to efficient code, often resulting in orders of magnitude better timing performance and memory footprint for both shared and distributed memory applications compared to state of the art approaches.

References

[1] E. A. Lee, “The problem with threads,” Computer, vol. 39, no. 5, pp. 33–42, 2006.

[2] G. Berry and L. Cosserat, “The synchronous programming language Esterel and its mathematical semantics,” in Seminar on Concurrency, 1984, vol. 197, pp. 389–448.

[3] D. M. Chapiro, “Globally-asynchronous locally-synchronous systems,” PhD Thesis, Stanford University, 1984.

[4] S. Ramesh, “Communicating reactive state machines: design, model and implementation,” in IFAC Workshop on Distributed Computer Control Systems, 1998.

[5] A. Malik, A. Girault, and Z. Salcic, “A GALS Language for Dynamic Distributed and Reactive Programs,” in ACSD, 2011.

[6] C. André, “SyncCharts: A visual representation of reactive behaviors,” Rapport de recherche tr95-52, Université de Nice-Sophia Antipolis, 1995.

[7] C. A. R. Hoare, “Communicating sequential processes,” Communications of the ACM, vol. 21, no. 8, pp. 666–677, 1978.

[8] W.-T. Sun, Z. Salcic, and A. Malik, “LibGALS: a library for GALS systems design and modeling,” in Proceedings of the 2010 Asia and South Pacific Design Automation Conference, Piscataway, NJ, USA, 2010, pp. 107–112.

[9] libDGALS web site, https://sites.google.com/site/libdgals/.

[10] G. Berry, S. Ramesh, and R. K. Shyamasundar, “Communicating reactive processes,” in POPL, 1993, pp. 85–98.

[11] W. D. Clinger, “Foundations of actor semantics,” 1981.

[12] M. Astley, The Actor Foundry. University of Illinois. 1999.

[13] J. Armstrong, R. Virding, C. Wikström, and M. Williams, “Concurrent programming in ERLANG,” 1993.

[14] P. Haller and M. Odersky, “Scala Actors: Unifying thread-based and event-based programming,” Theoretical Computer Science, vol. 410, no. 2–3, pp. 202–220, Feb. 2009.

[15] F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi, “JADE - a java agent development framework,” Multi-Agent Programming, pp. 125–147, 2005.

[16] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, “X10: an object-oriented

approach to non-uniform cluster computing," in OOPSLA, 2005, pp. 519–538.
[17] J. Galletly, Occam 2. Taylor & Francis, 1990.
[18] Microsoft Corporation, "Axum programming language." Sep-2008.
[19] W. Gropp, E. Lusk, and A. Skjellum, "Using MPI: portable parallel programming with the message passing interface," 1999.
[20] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," Computational Science & Engineering, IEEE, vol. 5, no. 1, pp. 46–55, 2002.
[21] R. Milner, Communicating and mobile systems: the pi calculus. Cambridge university press, 1999.