LeGESD with a Process Algebra Approach for the Specification of Distributed Systems

Jorge Cortes Galicia and Felipe Rolando Menchaca Garcia
Centro de Investigacion en Computacion
Instituto Politecnico Nacional
Av. Juan de Dios Batiz s/n Esq. Miguel Othon de Mendizabal, Mexico
jcortes@sagitario.cic.ipn.mx, fmenchac@ipn.mx

Abstract

Actually, significant progress has been made in the development of process algebras for the specification and analysis of distributed systems. This paper describes a process algebraic approach for LeGESD, a graphical specification language for distributed systems. LeGESD is a formal language for the specification and analysis of distributed systems including their functional and communication requirements. The process algebraic approach of LeGESD is called Analysis and Design of Distributed Systems (ADSD), an algebraic specification with operational semantic. ADSD is the semantic base of LeGESD and provides behavioral equivalence relations which can be used to verify the correctness of a LeGESD specification.

1. Introduction

The development of distributed systems uses techniques and tools for specification, design and code generation. We have presented LeGESD [3] for the specification of such systems, a graphical language used to specify easily distributed systems. Behind of any specification language is convenient that exist a formalism that support to the language. This formal base is useful to support reasoning about specifications and designs [9]. Although some formal methods for distributed systems have been developed using textual or graphical notations, such as Macedon [11], P2 [12], statecharts, TLA [8], IOA [4], LfP [7, 10] and PEDS [13]. However, some of them not present a formalims that support to the method.

This has motivated to development a process algebra [2, 5] approach for LeGESD such as alternative solution to the specification and analysis of distributed systems using an algebraic method. LeGESD's algebra process has two motivation: first motivation is to provide a formalism which helps to fast verification of distributed systems, this formalism should be consistent and well-defined.

The second motivation is to provide a solid operational semantics [6]. This operational semantics is defined by the Analysis and Design of Distributed Systems (ADSD), ADSD is a specification in algebraic level. In addition, ADSD has defined an operational semantics and equivalence translations. ADSD allows to specify an equivalence translation between a graphical notation in LeGESD to a textual notation in ADSD. This equivalence translation is used in the design of a distributed system and it will be useful in stages of model verification and source code generation.

2. LeGESD

The LeGESD language is based on the view that a distributed system consists of two levels of components: communication components, called *medio*, that use a finite set of serially communicated resources for communication execution and that synchronize with one another; behavior components, called *jobs*, that use a finite set of serial states for behavior execution. The description of a job is represented by *behavior diagrams* which use a set of graphical elements [3]. Exist two several graphical elements: *states* and *links*. The execution of a state is assumed to take a set of preconditions, and can generate a set of post-conditions once finished the execution. The execution of a state is subject to verification of the preconditions that it uses. On the other hand, the description of a medio is represented by *communication diagrams* which use a set of graphical elements [3]. This set is divided into two subsets: *initialization* and *execution*. Initialization subset refers to initial communication activities such as open, bind and begin that are common activities in all distributed communication technologies. Execution subset refers to communication internal behavior. These subsets use same several graphical elements like a

job (states and links), but define other symbols to represent states.

2.1. LeGESD jobs and medio

A LeGESD job is a tuple (S, I, L, Pr, Po, M, R) where Pr is a set of preconditions, Po is a set of post-conditions, M is a set of messages, and R is a set of medio references. (S, L) is a directed graph with initial node $I \subseteq S$. The set of labeled links $L \subseteq (S \times l \times S)$ is defined over the set of labels $l \subseteq \{\epsilon\} \cup (\mathcal{N} \times Pr) \cup (\mathcal{N} \times Po) \cup (M \times R)$ where ϵ denotes an empty label. Since LeGESD jobs is hierarchical, the set of states S is defined together with a *hierarchy function* Ψ which defines for each state the set of LeGESD jobs it contains inside. That is, $\Psi(s) = \{J_1, \cdots, J_k\}$ if the LeGESD jobs $J_1, \cdots, J_k$ are contained immediately inside the state s. Given the hierarchy function Ψ, a state s is either a begin, internal, communication, access point, end, or transition state, if $\Psi(s) = 0$; and a state s is a compound state, if $\Psi(s) = \{J_1, \cdots, J_k\}$.

On the other hand, the set of elements R is defined by a *compound function* Λ which defines for each medium the two LeGESD medio it contains inside. That is, $\Lambda(r) = \{I_m, E_m\}$ if the initialization LeGESD medio I_m and the execution LeGESD medio E_m are contained in the medio r. Given the compound function Λ, a not valid medio r is if $\Lambda(r) = 0$. An I_m is a tuple (S, B, L, Pr, Po, N) where Pr is a set of preconditions, Po is a set of post-conditions, and N is a set of medio names. (S, L) is a directed graph with initial node $B \subseteq S$. The set of labeled links $L \subseteq (S \times l \times S)$ is defined over the set of labels $l \subseteq \{\epsilon\} \cup (\mathcal{N} \times Pr) \cup (\mathcal{N} \times Po) \cup \{N\}$ where ϵ denotes an empty label. The set of states S is defined by a begin, open, bind, initial, end, access point, or transition state.

An E_m is defined in similar way to a job. E_m is a tuple (S, B, L, Pr, Po, M, N) where Pr is a set of preconditions, Po is a set of post-conditions, M is a set of messages, and N is a set of medio names. (S, L) is a directed graph with initial node $B \subseteq S$. The set of labeled links $L \subseteq (S \times l \times S)$ is defined over the set of labels $l \subseteq \{\epsilon\} \cup (\mathcal{N} \times Pr) \cup (\mathcal{N} \times Po) \cup (M \times N)$ where ϵ denotes an empty label. Since E_m is hierarchical, the set of states S is defined together with a *hierarchy function* Ω which defines for each state the set of execution LeGESD medio it contains inside. That is, $\Omega(s) = \{E_{m1}, \cdots, E_{mk}\}$ if the execution LeGESD medio $E_{m1}, \cdots, E_{mk}$ are contained immediately inside the state s. Given the hierarchy function Ω, a state s is either a begin, simple, access point, end, or transition state, if $\Omega(s) = 0$; and a state s is a compound state, if $\Omega(s) = \{E_{m1}, \cdots, E_{mk}\}$.

To assign precise semantics to LeGESD specification we next describe rules that restrict how LeGESD states and links can be combined to form a valid LeGESD specification.

Definition 1. A valid LeGESD job has links and states that satisfy the following restrictions:

Links.-

Each link connects states only at the same job.

For every link (s, l, k), and for every compound state s, $\Psi(s) = \{J_1, \cdots, J_k\}$, s is a state of J_i if and only if k is a state of J_i, for $i = 1, \cdots, k$.

States.-

Each begin state must have only labeled or unlabeled simple out-links and nothing else.

Each internal state can have only unlabeled or labeled simple out-links or in-links. It can have multiple unlabeled or labeled simple out-links or in-links.

Each communication state must have one labeled incoming or out-coming message link, also can have unlabeled or labeled simple out-links or in-links. A communication state has compound state characteristics, i.e., it describes the details of communication activities.

Each compound state can have unlabeled or labeled simple in-links, also can have labeled incoming or out-coming message link. It can have multiple unlabeled or labeled simple in-links.

Each end state is a sink state, i.e., a state with no out-links.

Each transition state can have only unlabeled or labeled simple out-links or in-links, and it can not be a sink state. If labeled simple in-links are presented will be necessary to verify the label (preconditions) to advance next state.

Each access point state must have only one labeled incoming or out-coming message link.

Definition 2. A valid initialization LeGESD medio has links and states that satisfy the following restrictions:

Links.-

Each link connects states only at the same initialization medio.

States.-

Each begin state must have only labeled or unlabeled simple out-links and nothing else.

Each open state can have only unlabeled or labeled simple out-links or in-links.

Each bind state can have only unlabeled or labeled simple out-links or in-links.

Each initial state must have labeled incoming or out-coming message links, also can have unlabeled or labeled simple in-links. It can have multiple labeled incoming or out-coming message links.

Each end state is a sink state, i.e., a state with no out-links.

Each transition state can have only unlabeled or labeled simple out-links or in-links, and it can not be a sink state. If labeled simple in-links are presented will be necessary to verify the label (preconditions) to advance next state.

Each access point state must have only one labeled incoming or out-coming message link.

Definition 3. A valid execution LeGESD medio has links and states that satisfy the following restrictions:

Links.-

Each link connects states only at the same execution medio.

For every link (m, l, q)*, and for every compound state m,* $\Omega(m) = \{E_{m1}, \cdots, E_{mk}\}$*, m is a state of* E_{mi} *if and only if q is a state of* E_{mi}*, for* $i = 1, \cdots, k$*.*

States.-

Each begin state must have only labeled or unlabeled simple out-links and nothing else.

Each simple state can have labeled incoming or out-coming message link, also can have unlabeled or labeled simple out-links or in-links. It can have multiple unlabeled or labeled simple out-links or in-links.

Each compound state can have unlabeled or labeled simple in-links, also can have labeled incoming or out-coming message link. It can have multiple unlabeled or labeled simple in-links.

Each end state is a sink state, i.e., a state with no out-links.

Each transition state can have only unlabeled or labeled simple out-links or in-links, and it can not be a sink state. If labeled simple in-links are presented will be necessary to verify the label (preconditions) to advance next state.

Each access point state must have only one labeled incoming or out-coming message link.

The underlying semantics of LeGESD is the Analysis and Design of Distributed Systems (ADSD).

3 LeGESD semantics

As we mentioned in the introduction, the underlying semantics of LeGESD is the Analysis and Design of Distributed Systems (ADSD). We will define the semantics of LeGESD in two steps. Before covering the details of the formal semantics of LeGESD, we first describe it informally. We then review ADSD and present some translations from LeGESD to ADSD. We finally describe the analysis techniques that are supported in the LeGESD formalism.

3.1 Informal semantics

Intuitively, a set of LeGESD jobs and medio represents a system. A system can execute executions and communication events. In LeGESD, this is described through the LeGESD job states and LeGESD medio states, respectively. In addition to this basic notion of execution, several jobs and medio can be combined through compound states (both media and jobs) in LeGESD to describe a larger system where jobs execute in sequence.

3.2 ADSD formal semantics

The underlying semantics of LeGESD is the Analysis and Design of Distributed Systems (ADSD). We define the semantics of LeGESD in two steps. The first step is to identify *basic* LeGESD elements, a subset of LeGESD (jobs or media), with an obvious correspondence with ADSD. The second step is to define a set of transformations from basic LeGESD elements to LeGESD and viceversa. Thus, every valid LeGESD job or medio can be translated into an ADSD job or medio by first converting it to a basic LeGESD job or medio and then mapping the simple LeGESD job or medio to a corresponding ADSD job or medio.

ADSD augments CCS [5] with discrete time execution actions. ADSD has one type of actions: execution actions, called *events*. An event in ADSD consists of one element *(e)* where *e* is a label. Let *P* range over the domain of terms, *e* range over the domain of event labels or *t*, *F* range over the set of event labels, and let *X* range over the domain of term variables. ADSD's syntax is given by the following grammar:

$P ::= NIL \mid (e).P \mid P_1 + P_2 \mid P_1 \parallel P_2 \mid P\Delta^b(P_1) \mid P/F \mid rec\ X.P \mid X$

The semantics of ADSD processes is defined in terms of labeled transition system together with a notion of prioritized transition. A transition is denoted as $P \longrightarrow^{\alpha} P'$ for processes *P* and *P'* and an action α that *P* can execute. Due to space limitation, we only informally describe the ADSD semantics.

NIL is a process that executes no action, e.g, it is initially deadlocked. There are one prefix operator that correspond to the one type of action. *(e).P*, executes the event *e* and proceeds to *P*. The Choice operator $P_1 + P_2$ represents possibilities - either of the processes may be chosen to execute, subject to the event offerings and resource limitations of the environment. The operator $P_1 \parallel P_2$ is the concurrent execution of P_1 and P_2.

The Scope construct $P\Delta^b(P_1)$ binds the process *P* by a event scope. The scope may be exited in one way. If *P* successfully terminates by executing an event labeled with *b*, then control proceeds to the "success-handler" P_1 (here, *b* may be any label other than *t*).

The Restriction operator, *P/F*, limits the behavior of *P*: events with labels in *F* are permitted to execute only if they synchronize and become the internal event *t*. The process *rec X.P* denotes standard recursion, allowing the specification of infinite behaviors. The term *X*, without a *"rec"* binding, is a free variable that belongs to an infinite set of free term variables.

ADSD offers two basic notions of behavioral equivalence that are defined over the prioritized labeled transition

system. The first equivalence relation is based on strong bisimulation [13], π, which ensures that equivalent processes match each other's labeled transitions. The second is based on weak bisimulation, Π, which ensures that equivalent processes match each other's transitions that are labeled with non-*t* events but allows one process to make transitions on *t* that an equivalent process does not match.

3.3 LeGESD to ADSD translation

In this section, we define the semantics of a subset of valid LeGESD, called *basic* LeGESD. Basic LeGESD corresponds to ADSD; that is, there is a translation between basic LeGESD to ADSD and vice versa. Due to space limitation, we only describe the first translation.

To define the semantics of LeGESD, we connect basic LeGESD with LeGESD via a set of graphical transformations that are sound with respect to the prioritized strong equivalence, π, of ADSD. The graphical transformations basically eliminate or add links and states to convert a LeGESD job to a basic LeGESD job or vice versa. In addition to connecting LeGESD with basic LeGESD, the graphical transformations allow to minimize a LeGESD specification by removing unnecessary states and links and producing a more succinct or simple LeGESD specification that is equivalent to the original one.

A *basic* LeGESD job is a valid LeGESD such that:

1. Any transition state has either one simple labeled out link, or one/two simple unlabeled/labeled out link;
2. Any compound state is a sink state;
3. Any compound state, s, has either one or two nested components, i.e., p(s) = G or p(s) = G1, G2; and
4. Any compound state with out links has one nested component and one simple unlabeled link.

Each basic LeGESD job corresponds to an ADSD process. The translation of a LeGESD job consists of translating the states and the links. The translation starts from the initial state and is recursively applied to jobs reachable from the initial states. The results of the translations are then combined using an ADSD operator. Figure 1 graphically describes the eight steps for translating a LeGESD job to an ADSD process, where *T* represents a translation from LeGESD specification to ADSD processes. Theses steps are resuming below:

Step 1 binds the translation of the LeGESD job to the process variable name P. In step 2, an event prefix process is created from the event that labels the simple link out of the transition state. In step 3, each job that starts at the target state of the unlabeled link is translated and the result is combined through the Choice operator. In step 4, the end state

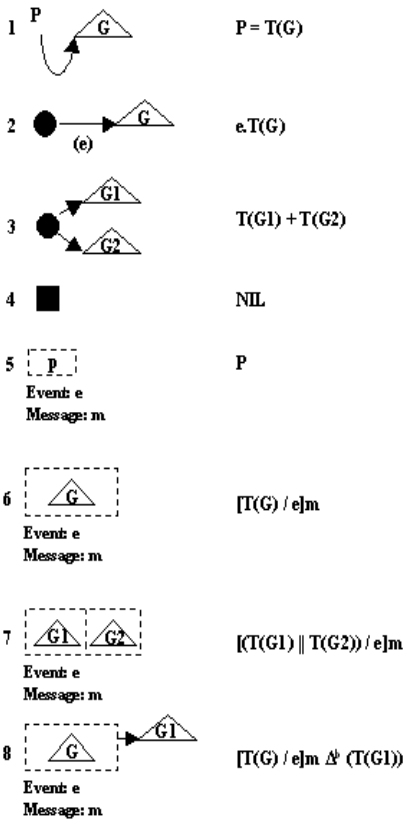


Figure 1. Basic LeGESD to ADSD translation

is translated to the NIL process. In step 5, the compound state is translated to a process variable with the referenced name. In step 6, the LeGESD job inside the compound state is translated, and then the Event and Message attributes of the compound state are used in the Restrict and Close operators; the attributes are ignored if they are the empty sets. In step 7, the two LeGESD jobs inside the compound state are translated and combined through the Parallel operator. The Event and Message attributes of the compound state are used in the Restrict and Close operators. In step 8, the LeGESD job inside the compound state is translated and used as the main process in the Scope operator. The translation of the LeGESD job that starts at the target state of the unlabeled link produces the interrupt process.

We can use structural induction to prove that for every basic LeGESD job *G*, *T(G)* is an ADSD process and that for every ADSD process *P*, there is a basic LeGESD job *G* such that *T(G)* is *P*. Furthermore, we can establish the uniqueness of the correspondence between basic LeGESD and ADSD. We omit the details due to space limitation.

4. A simple translation example

This section illustrates on a short example the translation between LeGESD and ADSD. The example is trivial but useful to illustrate the translation basis. We just present a few parts of this translation. The distributed system to specify is presented in Figure 2, it is a distributed calculator where four basic arithmetic operations are distributed. Figure 2 presents both the System view and the Deployment view of LeGESD. System view presents several compound states of LeGESD job (Calculator, Add, Sub, Div, Mult) and LeGESD medio (Calculator_medio, Add_medio, Sub_medio, Div_medio, Mult_medio). Deployment view presents constraint and deployment conditions with global specifications about initial constructions requirements (management information and formal declarations). Both System and Deployment view represent the highest specification level of LeGESD about the system to specify. Different compound states in the System view have to be represented with a respective LeGESD job. Meanwhile communication states have to be represented with a respective LeGESD medio if the corresponding access point exist.

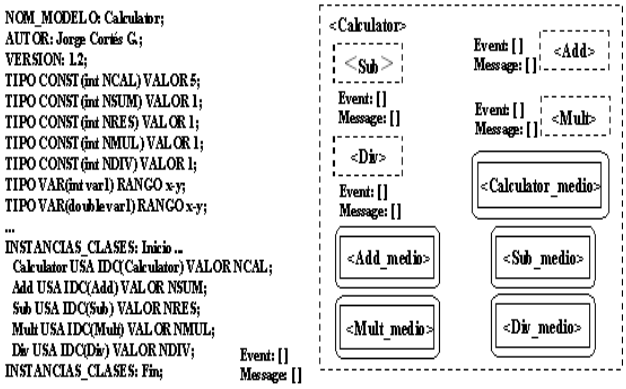


Figure 2. System and Deployment view of LeGESD

In Figure 2, we can view that global compound state Calculator does not a basic LeGESD job. However inside of this global compound state exists some basic LeGESD jobs (Add, Sub, Div, Mult) according to the number 3 basic LeGESD job condition defined in subsection 3.3. Thus each basic LeGESD job corresponds to an ADSD process, the translations are presented for each basic LeGESD job inside of Calculator compound state in figure 3.

Figure 4 presents a version of LeGESD job for Calculator compound state. This figure presents the behavior diagram of Calculator LeGESD job. It shows the execution workflow of Calculator job. Communication states (Operation and Wait) are linked through transitions states and labeled and unlabeled simple links which connect to the job states. We have defined two access points (Calculator_medio and Add_medio) to make an add operation binding to the medium. These access points are linked to communication states through labeled incoming and out-coming message links. We specify all necessary labels for each link with the attributes corresponding to preconditions and post-conditions. Labeled incoming and out-coming message links contain messages to interchange with the medio through each access point.

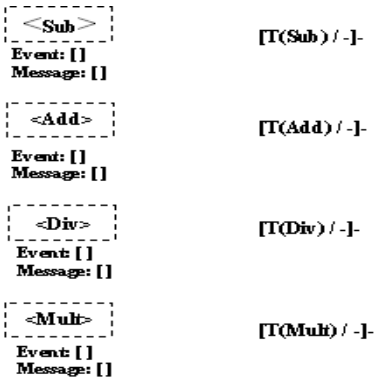
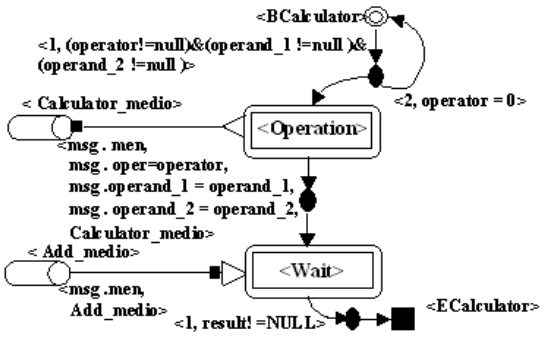


Figure 3. Translation from basic LeGESD job to ADSD



Calculator = {T(Operation) + T(BCalculator), T(Wait), T(ECalculator),
[T(Operation) / -]msg Δ^b (T(Calculator_medio)),
[T(Wait) / -]msg Δ^b (T(Add_medio))}

Figure 4. Calculator LeGESD job diagram

The behavior diagram shows a basic LeGESD job according to the number 1 and 2 basic LeGESD job condition defined in subsection 3.3. Thus basic LeGESD job corresponds to an ADSD process, the translations are presented in the figure. The first translation uses Choice operator. We

also show translation from LeGESD medio state to ADSD using step 8 for Scope operator, msg represents the message send or receive to or from media.

5. Conclusion

We presented the semantic behind LeGESD language (ADSD) for the specification of distributed systems. ADSD is offering a process algebra approach to have a precise semantic suitable for the description of interaction between LeGESD jobs and LeGESD medio in a distributed system. Based on the study presented in this paper, ADSD has some capabilities for verification of distributed systems models built with LeGESD. ADSD also allows the verification of complex systems supporting the modular and hierarchical fashion of LeGESD, it allows the integration of the communication requirements of a system with its behavior requirements; ADSD has a precise process algebraic semantics which can support analysis through execution and formal verification of the specification. We note that ADSD is compatible with the notion of modularity in LeGESD allows designers to specify and to verify a system through its components incrementally.

Acknowledgement: We thanks to the "Instituto Politecnico Nacional", "Centro de Investigacion en Computacion" and "Escuela Superior de Computo" for their support in this paper.

References

[1] M. Bravetti and M. Bernardo. Compositional asymmetric cooperations for process algebras with probabilities, priorities and time. *1st International Workshop on Models for Time Critical Systems*, 39, 2000.

[2] J. Cortes and F. R. Menchaca. Algebra de procesos aplicada a la especificacion formal de sistemas distribuidos. *1er. Congreso Internacional en Sistemas Computacionales y Electrnicos (CISCE 06)*, pages 30–38, 2006.

[3] J. Cortes and F. R. Menchaca. Graphical specification language for distributed systems. *Proceeding IEEE on 15th International Conference on Computing*, pages 120–126, 2006.

[4] S. J. Garland and N. A. Lynch. *The IOA language and toolset: Support for designing, analyzing, and building distributed systems*. MIT Press. Technical Report MIT/LCS/TR-762, Cambridge, MA, 1998.

[5] H. Hermanns and U. Herzog. *Process algebra for performance evaluation*. Theoretical Computer Science, 2001.

[6] K. Honda and K. Tokoro. On asynchronous communication semantics. *Object-Based Concurrent Computing*, 612:21–51, 1992.

[7] F. Kordon and I. Mounier. Formal verification of embedded distributed systems in a prototyping approach. *Workshop on Engineering Automation for Software Intensive System Integration*, 2001.

[8] L. Lamport. *Introduction to TLA*. Digital SRC Technical Note, 1994.

[9] D. C. Luckham. Specification and analysis of system architecture. *IEEE Transactions on Software Engineering*, (21):336–355, 1995.

[10] D. Regep and F. Kordon. Lfp: A specification language for rapid prototyping of concurrent systems. *Proceedings of the 12th International IEEE Workshop on Rapid System Prototyping*, pages 90–97, 2001.

[11] A. Rodriguez and C. Killian. Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. *Proceedings of the NSDI*, 2004.

[12] B. Thau Loo and T. Condle. Implementing declarative overlays. *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005.

[13] D. Zhang and K. Zhang. A visual programming environment for distributed systems. *Proceedings of the 11th International IEEE Symposium on Visual Languages*, pages 310–317, 1995.