A Scalable Approach to the Design of SW Architectures with Dynamically Created/Destroyed Components

Pietro Abate
Australian National University
RSISE, Automatic Reasoning Group, Canberra
abate@arp.anu.edu.au

Marco Bernardo
Università di Urbino
Centro per l'Appl. delle Scienze e Tecnol. dell'Inf.
bernardo@sti.uniurb.it

ABSTRACT

The architecture of component based software systems is classified as being static or dynamic, depending on whether the component number and the component connections are fixed a priori or can change at run time. Most work in the field of formal method based architectural description languages has focused on static architectures, as well as dynamic architectures where the architectural specification does not scale with respect to the components that can be created or destroyed at run time. In this paper we start from PADL, a graphical, hierarchical, process algebra based language for the description of static software architectures. We then enrich its syntax and semantics in order to provide scalable specifications of software architectures where some components are dynamically created and destroyed. We show that the construction of the new language allows the architectural checks developed for PADL to be reused for the detection of architectural mismatches in the description of dynamic software architectures.

1. INTRODUCTION

Architecture based software development is an approach to software design focusing on high level models of software systems rather than on program source code [9, 10]. In this context, architectural description languages (ADLs) have been developed to support software architecture design and analysis. ADLs force the developers to model systems in a more controlled way, which in particular elucidates the basic architectural concepts of component and connection, thus hopefully enhancing software reuse and maintainability, and permits the detection of architectural mismatches whenever the ADLs are based on formal methods [2, 3].

The architecture of component based software systems is classified as being static or dynamic, depending on whether the component number and the component connections are fixed a priori or can change at run time. Complex software systems can require that their architecture be modified at run time in different ways, such as addition of new components, upgrade of existing components, deletion of components, and reconfiguration of component connections.

Most work in the field of formal method based ADLs has focused on static architectures, as well as dynamic architectures where the architectural specification does not scale w.r.t. the components that can be created/destroyed at run time. The purpose of this paper is to enable a scalable design of software architectures where some components are dynamically created and destroyed. In our work we start with a language for the description of static software architectures. To this aim we choose PADL [3], a graphical, hierarchical, process algebra based ADL equipped with a set of compatibility, interoperability and conformity checks at the architectural level based on behavioral equivalences. Then we enrich the syntax and the semantics for PADL in order to be able to specify dynamic software architectures in a scalable way. More precisely, the new ADL, called ΔPADL, permits to describe software architectures with some components dynamically created and destroyed, where each description is parameterized w.r.t. the maximum number of dynamic components of each type and simply requires the provision of the connections for a prototype dynamic component of each type. The strength of ΔPADL is that its specifications, besides scaling w.r.t. the maximum number of dynamic components of each type, make the component creation/destruction related details transparent to the architect and can undergo to the same architectural checks as PADL specifications. This is achieved by automatically translating every ΔPADL specification into a static PADL specification in which the declared maximum number of dynamic components of each type is present in an activated/deactivated status, with some transparent instance managers and routers taking care of the implementation of the dynamism.

This paper is organized as follows. In Sect. 2 we recall the basic concepts about PADL. In Sect. 3 we define the syntax and the translation semantics for ΔPADL. In Sect. 4 we present an application of ΔPADL to a distributed leader election algorithm. Finally, in Sect. 5 we report some concluding remarks about related and future work.

2. PADL

In this section we recall from [3] the syntax, the semantics, and the architectural checks for PADL, an ADL for the compositional, graphical, and hierarchical modeling of software systems. PADL is based on a process algebra called PA. The set of process terms of PA is generated by the following syntax

$$E ::= \underline{0} \mid a.E \mid E/L \mid E[\varphi] \mid E + E \mid E \parallel_S E \mid A$$

SEKE 2002 Ischia, Italy EU

where a belongs to a set Act of actions including a distinguished action τ for unobservable activities, $L, S \subseteq Act - \{\tau\}$, φ belongs to a set $ARFun$ of action relabeling functions preserving observability (i.e., $\varphi^{-1}(\tau) = \{\tau\}$), and A belongs to a set $Const$ of constants each possessing a (possibly recursive) defining equation of the form $A \triangleq E$. In the syntax above, $\underline{0}$ is the term that cannot execute any action. Term $a.E$ can execute action a and then behaves as term E. Term E/L behaves as term E with each executed action a turned into τ whenever $a \in L$. Term $E[\varphi]$ behaves as term E with each executed action a turned into $\varphi(a)$. Term $E_1 + E_2$ behaves as either term E_1 or term E_2 depending on whether an action of E_1 or an action of E_2 is executed. Term $E_1 \parallel_S E_2$ asynchronously executes actions of E_1 or E_2 not belonging to S and synchronously executes equal actions of E_1 and E_2 belonging to S. The action prefix operator and the alternative composition operator are called dynamic operators, whereas the hiding operator, the relabeling operator, and the parallel composition operator are called static operators. A term is called sequential if it is composed of dynamic operators only. The semantics for PA is defined through inference rules that map every guarded and closed term E to a state transition graph $\mathcal{I}[\![E]\!]$, whose states are represented by process terms and whose transitions are labeled with actions. The notion of equivalence that we consider for PA is the weak bisimulation equivalence [8], denoted $\approx_{\mathrm{B}}$, which captures the ability of two terms to simulate each other behaviors up to τ actions.

A description in PADL represents an architectural type. Each architectural type is defined as a function of its architectural element types (AETs) and its architectural topology. An AET is defined as a function of its behavior, specified either as a family of sequential PA terms or through an invocation of a previously defined architectural type, and its interactions, specified as a set of PA actions occurring in the behavior that act as interfaces for the AET. The architectural topology is specified through the declaration of a set of architectural element instances (AEIs) representing the system components, a set of architectural (as opposed to local) interactions given by some interactions of the AEIs that act as interfaces for the whole architectural type, and a set of directed architectural attachments among the interactions of the AEIs. Every interaction is declared to be an input interaction or an output interaction and the attachments must respect such a classification: every attachment must involve an output interaction and an input interaction of two different AEIs. An AEI can have different types of interactions (input/output, local/architectural); it must have at least one local interaction. Every local interaction must be involved in at least one attachment, while every architectural interaction must not be involved in any attachment. In order to allow several AEIs to synchronize, every interaction can be involved in several attachments provided that no autosynchronization arises, i.e. no chain of attachments is created that starts from an interaction of an AEI and terminates on an interaction of the same AEI.

We show in Table 1 a PADL textual description for a pipe-filter system. The system is composed of three identical filters and one pipe. Each filter acts as a service center of capacity two that is subject to failures and subsequent repairs. For each item processed by the upstream filter, the pipe forwards it to one of the two downstream filters according to the availability of free positions in their buffers. If both have free positions, the choice is resolved nondeterministically. The same system is depicted in Fig. 1 through the PADL graphical notation, which is based on flow graphs [8]. In a flow graph, the boxes denote the AEIs, the black circles denote the local interactions, the white squares denote the architectural interactions, and the directed edges denote the attachments.

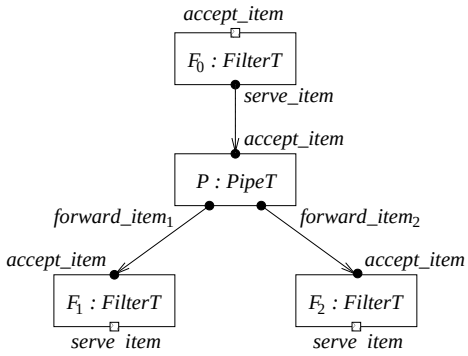

Figure 1: Graphical description of *PipeFilter*

The semantics of a PADL specification is given by translation into PA in two steps. In the first step, the semantics of all the instances of each AET is defined to be the behavior of the AET projected onto its interactions. Such a projected behavior is obtained from the family of sequential PA terms representing the behavior of the AET by applying a hiding operator on all the actions that are not interactions. In this way, we abstract from all the internal details of the behavior of the instances of the AET. For the pipe-filter example

$$\begin{array}{lll} [\![FilterT]\!] = & [\![F_0]\!] = [\![F_1]\!] = [\![F_2]\!] & = Filter/\{fail, repair\} \\ [\![PipeT]\!] = & [\![P]\!] & = Pipe \end{array}$$

In the second step, the semantics of an architectural type is obtained by composing in parallel the semantics of its AEIs according to the specified attachments, possibly after relabeling to the same action the interactions that are involved in the same chain of attachments. For the pipe-filter example

$$\begin{array}{rcl} [\![PipeFilter]\!] & = & [\![F_0]\!][serve\_item \mapsto a] \parallel_{\emptyset} \\ & & \quad [\![F_1]\!][accept\_item \mapsto a_1] \parallel_{\emptyset} \\ & & \quad\quad [\![F_2]\!][accept\_item \mapsto a_2] \parallel_{\{a,a_1,a_2\}} \\ & & \quad\quad\quad [\![P]\!][accept\_item \mapsto a, \\ & & \quad\quad\quad\quad forward\_item_1 \mapsto a_1, \\ & & \quad\quad\quad\quad forward\_item_2 \mapsto a_2] \end{array}$$

PADL is equipped with two $\approx_{\mathrm{B}}$ based architectural checks that the designer can use to verify the well formedness of the architectural types and, in case a mismatch is detected, to identify the components that cause it. Such checks take care of verifying whether the deadlock free AEIs of an architectural type fit together well, i.e. do not lead to system blocks. The first check (compatibility) is concerned with architectural types whose topology is acyclic. For an acyclic architectural type, if we take an AEI K and we consider all the AEIs $C_1, \ldots, C_n$ attached to it, we can observe that they form a star topology whose center is K, as the absence of cycles prevents any two AEIs among $C_1, \ldots, C_n$ from communicating via an AEI different from K. It can easily be recognized that an acyclic architectural type is just a composition of star topologies. A $\approx_{\mathrm{B}}$ based compatibility check ensures the absence of deadlock within a star topology whose

archi_type	*PipeFilter*
archi_elem_types	
elem_type	*FilterT*
behavior	$Filter \triangleq accept\_item.Filter' + fail.repair.Filter$
	$Filter' \triangleq accept\_item.Filter'' + serve\_item.Filter + fail.repair.Filter'$
	$Filter'' \triangleq serve\_item.Filter' + fail.repair.Filter''$
interactions	**input** *accept_item*
	output *serve_item*
elem_type	*PipeT*
behavior	$Pipe \triangleq accept\_item.(forward\_item_1.Pipe + forward\_item_2.Pipe)$
interactions	**input** *accept_item*
	output $forward\_item_1, forward\_item_2$
archi_topology	
archi_elem_instances	$F_0, F_1, F_2 : FilterT$
	$P : PipeT$
archi_interactions	**input** $F_0.accept\_item$
	output $F_1.serve\_item, F_2.serve\_item$
archi_attachments	**from** $F_0.serve\_item$ **to** $P.accept\_item$
	from $P.forward\_item_1$ **to** $F_1.accept\_item$
	from $P.forward\_item_2$ **to** $F_2.accept\_item$
end	

Table 1: Textual description of *PipeFilter*

center K is deadlock free, and this check scales to the whole acyclic architectural type. The basic condition to check is that every C_i is compatible with K, i.e. the parallel composition of their semantics is weakly bisimulation equivalent to the semantics of K itself. Intuitively, this means that attaching C_i to K does not alter the behavior of K, i.e. K is designed in such a way that it suitably coordinates with C_i. Since the compatibility check is not sufficient for cyclic architectural types, the second check (interoperability) deals with cycles. A $\approx_B$ based interoperability check ensures the absence of deadlock within a cycle $C_1, \ldots, C_n$ of AEIs in the case that at least one of such AEIs is deadlock free. The basic condition to check is that at least one deadlock free C_i interoperates with the other AEIs in the cycle, i.e. the parallel composition of the semantics of the AEIs in the cycle projected on the interactions with C_i only is weakly bisimulation equivalent to the semantics of C_i. Intuitively, this means that inserting C_i into the cycle does not alter the behavior of C_i, i.e. that the behavior of the cycle assumed by C_i matches the actual behavior of the cycle.

A PADL description represents a family of software architectures called an architectural type. An instance of an architectural type can be obtained by invoking the architectural type and passing actual AETs and actual names for the architectural interactions. If each actual AET conforms to the corresponding formal AET, i.e. the semantics of the two AETs are weakly bisimulation equivalent up to an injective relabeling, then the semantics of the architectural type invocation and the architectural type definition are weakly bisimulation equivalent up to relabeling. As a consequence, all the conformant invocations of an architectural type possess the same compatibility and interoperability properties. The architectural type invocation mechanism, together with the possibility of defining architectural interactions, can be exploited to model system architectures in a hierarchical way.

3. ΔPADL

In this section we present the syntax and the semantics for an extension of PADL, called ΔPADL, that supports the description of architectural types whose number of AEIs can vary at run time through their dynamic creation and destruction. As we shall see, the main feature of ΔPADL is that its specifications scale w.r.t. the number of dynamic AEIs, with all the details related to the implementation of the dynamism being transparent to the designer. Another strength is that, despite the increased expressiveness, ΔPADL provides the architect with the same analysis capabilities as PADL, because the architectural compatibility, interoperability and conformity checks can be applied to ΔPADL specifications as well.

3.1 Dynamic AETs

The syntax for ΔPADL is the same as the syntax for PADL, with the additional capability of defining both static and dynamic AETs in the archi_elem_types section, as well as additional primitives for the dynamic creation/destruction of AEIs that will be explained later. A static AET (SAET) is an AET whose number of instances is fixed a priori in the archi_topology section. Its definition is preceded by the keyword static_elem_type and comprises a behavior and an interaction sections.

A dynamic AET (DAET) is an AET whose number of instances can vary at run time. Its definition is preceded by the keyword dynamic_elem_type and comprises a behavior, an interaction and a max_inst_no sections, with the last one establishing the maximum number of instances of the

DAET that can be simultaneously active at run time. In the case of a DAET, the maximum number of instances can be a parameter of the whole architectural type and the archi_topology section contains the declaration of a single prototype instance of the DAET with the related attachments. [1] It is intended that every instance of the DAET created at run time will have the same attachments as the declared prototype instance. In this way the specification of the architectural type is completely independent of the actual number of instances of the DAET. Later on we explain how the sharing of the same attachments by the active instances of the DAET is transparently managed through an identification mechanism and dynamic instance routers.

3.2 AEI Identification

In ΔPADL a mechanism for the identification of the AEIs is introduced. The reason is that, whenever an AEI wants to interact with an instance of a DAET, the former must specify with which instance of the DAET it wants to interact. Notice that this cannot be always established a priori, as the instances of the DAET that are active can vary from instant to instant at run time. Such an identification mechanism is implemented through the provision of a local environment to each AEI, in a way that is completely transparent to the designer. The local environment of an AEI is composed of the following integer valued variables: *self* contains the identifier of the AEI, which is a positive integer assigned upon creation; *sender* contains the identifier of the AEI that sent the last message intercepted by the AEI under consideration; *receiver* contains the identifier of the AEI to which the last message intercepted by the AEI under consideration is destinated.

In order to fully realize the identification mechanism, we need to introduce value passing capabilities in PA, i.e. the fact that actions can carry values across process terms. Following [8], we modify the syntax for PA as follows. First, besides unstructured actions, we allow for output actions of the form $a!(\underline{e})$, where $\underline{e}$ is a vector of expressions, and input actions of the form $a?(\underline{x})$, where $\underline{x}$ is a vector of variables. Second, a conditional operator of the form if (β) then E_1 else E_2, where β is a boolean expression, is introduced to allow data to drive the computation. Third, every constant is equipped with a vector of typed formal parameters $\underline{x}_1$ and local variables $\underline{x}_2$, thus resulting in constant defining equations of the form $A(\underline{x}_1;\underline{x}_2) \triangleq E$ where the corresponding invocations are of the form $A(\underline{e}_1)$. The resulting value passing process algebra, which we call $\mathrm{PA_{vp}}$, is used in ΔPADL to express the behavior of the AETs, with the understanding that output (input) actions must be declared as output (input) interactions. By so doing, an AEI can send a message to a specific instance of a DAET by simply including the receiver identifier among the vector of expressions of the corresponding output action. Moreover, if the behavior of an AET is given by the constant defining equation $A(\underline{x}_1;\underline{x}_2) \triangleq E$, in principle such an equation is transparently converted to

$$A(\underline{x}_1;\underline{x}_2, \mathrm{Id}\ \mathit{self}, \mathrm{Id}\ \mathit{sender}, \mathrm{Id}\ \mathit{receiver}) \triangleq E$$

in order to account for the identification mechanism related local environment, where Id is the type denoting the set of identifiers. As a consequence, every output action of the form $a!(id,\underline{e})$ is transparently converted to $a!(\mathit{self}, id, \underline{e})$ and every input action of the form $a?(\underline{x})$ is transparently converted to $a?(\mathit{sender}, \mathit{receiver}, \underline{x})$.

[1] Several prototype instances need to be declared whenever there is at least one attachment involving two of them.

3.3 Dynamic Primitives

The dynamic creation/destruction of the instances of the DAETs can be expressed within the behavior of an AEI through the two following dynamic primitives. An instance of DAET $\mathcal{C}$ can be created through the primitive given by the sequence of $\mathrm{PA_{vp}}$ actions $\mathit{create}_{\mathcal{C}}!(\mathit{self}).\mathit{get\_id}_{\mathcal{C}}?(\mathit{receiver}, id)$ where id is a local variable that will hold the identifier of the newly created instance. Instance id of DAET $\mathcal{C}$ can be destroyed through the primitive given by the $\mathrm{PA_{vp}}$ action $\mathit{destroy}_{\mathcal{C}}!(id)$. The actions composing the dynamic primitives are transparently added to the interactions of every AET that uses the dynamic primitives and attached to the corresponding interactions of transparent dynamic instance managers explained later on, which carry out the dynamic creation/destruction of the instances of the DAETs.

3.4 Dynamic Instance Manager and Routers

The semantics of a ΔPADL specification is given by translation into a static PADL specification relying on $\mathrm{PA_{vp}}$, so that the architectural compatibility, interoperability and conformity checks defined for PADL can be applied to ΔPADL as well. Such a translation is transparently accomplished by replacing every DAET $\mathcal{C}$ in the ΔPADL specification with an invocation to a PADL architectural type $\mathcal{A}_{\mathcal{C}}$ based on four SAETs as depicted in Fig. 2 (where names enclosed in angular parentheses stand for sets of uniquely indexed names) and by attaching the input/output related architectural interactions of $\mathcal{A}_{\mathcal{C}}$ in the same way as the interactions of the prototype instance of $\mathcal{C}$ would be.

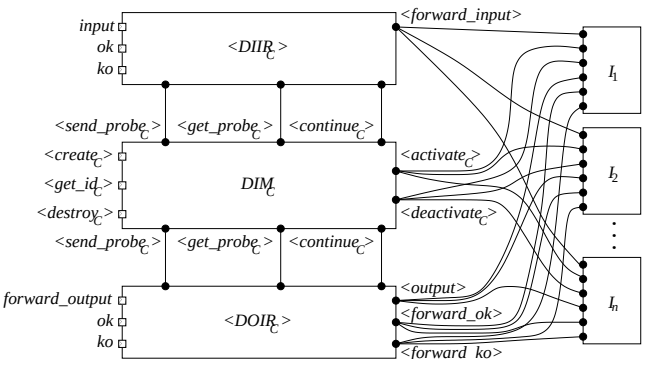

Figure 2: Flow graph of a DAET semantics

The first SAET has the same behavior and interactions as $\mathcal{C}$ itself. The number of instances ($I_1, I_2, \ldots, I_n$ in Fig. 2) of such a SAET declared in the archi_topology section is equal to the maximum number of instances of $\mathcal{C}$, with the instances having consecutive initial values for variable *self*. Every interaction of such instances is attached to the corresponding interaction of the other three SAETs (dynamic instance manager and routers) of $\mathcal{A}_{\mathcal{C}}$.

The second SAET is a dynamic instance manager called $\mathit{DIM}_{\mathcal{C}}$, which is responsible for carrying out the dynamic creation/destruction of the instances of $\mathcal{C}$ through the activation/deactivation of the corresponding instances of the first SAET. $\mathit{DIM}_{\mathcal{C}}$ has a single instance equipped with three sets of interactions with the outside for the requests of dynamic creation/destruction of the instances of $\mathcal{C}$ ($\mathit{create}_{\mathcal{C}}$,

$get\_id_{\mathcal{C}}$, $destroy_{\mathcal{C}}$), three sets of interactions for the atomic check for existence of the instances of $\mathcal{C}$ that can be requested by the dynamic instance routers ($send\_probe_{\mathcal{C}}$, $get\_probe_{\mathcal{C}}$, $continue_{\mathcal{C}}$), and two sets of interactions for the activation/deactivation of the instances of the first SAET ($activate_{\mathcal{C}}$, $deactivate_{\mathcal{C}}$). Every element in a set of interactions has a unique index and is attached to a single interaction of an AEI requiring a dynamic creation/destruction or check for existence or of an instance of the first SAET. In essence, $DIM_{\mathcal{C}}$ handles a table of the instances of $\mathcal{C}$, with value 0/1 in an entry indicating the activation/deactivation of the corresponding instance, depending on the requested services. The behavior of $DIM_{\mathcal{C}}$ is shown below; for the sake of simplicity, we show the behavior only for the element having index i of each set of interactions:

$DIMBeh_{\mathcal{C}}(\text{array}(max\_inst\_no,\text{int})\ act\_inst;\text{Id}\ sender,\text{Id}\ id) \triangleq$
  $create^i_{\mathcal{C}}?(sender).FindId^i_{\mathcal{C}}(act\_inst, sender, 0) +$
  $destroy^i_{\mathcal{C}}?(id).$
    if $(\text{read}(id, act\_inst) = 0)$ then
      $DIMBeh_{\mathcal{C}}(act\_inst)$
    else
      if $(id = 1)$ then
        $deactivate^1_{\mathcal{C}}.DIMBeh_{\mathcal{C}}(\text{write}(id, 0, act\_inst))$
      else if $(id = 2)$ then
        $deactivate^2_{\mathcal{C}}.DIMBeh_{\mathcal{C}}(\text{write}(id, 0, act\_inst))$
      else if ...
        ...
      else
        $deactivate^n_{\mathcal{C}}.DIMBeh_{\mathcal{C}}(\text{write}(id, 0, act\_inst)) +$
  $send\_probe^i_{\mathcal{C}}?(id).get\_probe^i_{\mathcal{C}}!(\text{read}(id, act\_inst)).$
    $continue^i_{\mathcal{C}}.DIMBeh_{\mathcal{C}}(act\_inst)$

$FindId^i_{\mathcal{C}}(\text{array}(max\_inst\_no,\text{int})\ act\_inst,\text{Id}\ sender,\text{Id}\ id;) \triangleq$
  if $(\text{read}(id, act\_inst) = 1)$ then
    $FindId^i_{\mathcal{C}}(act\_inst, id + 1)$
  else
    if $(id = 1)$ then
      $activate^1_{\mathcal{C}}.get\_id^i_{\mathcal{C}}!(sender, id).$
        $DIMBeh_{\mathcal{C}}(\text{write}(id, 1, act\_inst))$
    else if $(id = 2)$ then
      $activate^2_{\mathcal{C}}.get\_id^i_{\mathcal{C}}!(sender, id).$
        $DIMBeh_{\mathcal{C}}(\text{write}(id, 1, act\_inst))$
    else if ...
      ...
    else
      $activate^n_{\mathcal{C}}.get\_id^i_{\mathcal{C}}!(sender, id).$
        $DIMBeh_{\mathcal{C}}(\text{write}(id, 1, act\_inst))$

The activation mechanism requires that in the behavior of each instance j of the first SAET the first constant defining equation, say $A^j(\underline{x}_1;\underline{x}_2) \triangleq E$, be preceded by an additional constant defining equation of the form

$$A^j_{\mathcal{C}}(\underline{x}_1;) \triangleq activate^j_{\mathcal{C}}.A^j(\underline{x}_1)$$

in order to allow an instance to operate only after it has been activated. On the other hand, the deactivation mechanism requires that in the behavior of each instance j of the first SAET every subterm of the form $a.E$ be changed into

$$a.E + deactivate^j_{\mathcal{C}}.A^j_{\mathcal{C}}(\underline{e}_1)$$

in order to allow an instance to be deactivated at any instant. Finally, the check for existence is conducted in an atomic way via action $continue_{\mathcal{C}}$; $DM_{\mathcal{C}}$ cannot go on before receiving such a signal back from the dynamic instance router that requested the check. This avoids the situation in which an instance is checked to be activated by a dynamic instance router, then the instance is deactivated by an AEI, then the dynamic instance router fails to forward a message to the instance.

The third SAET is a dynamic input instance router called $DIIR_{\mathcal{C}}$, which handles the attachments shared by the input interactions of the instances of $\mathcal{C}$. $DIIR_{\mathcal{C}}$ has as many instances as there are input interactions, where each instance is equipped with the input interaction with the outside that it routes, as many output interactions as there are instances of $\mathcal{C}$ to forward the received messages ($forward\_input$), and two output interactions to send back the positive/negative acknowledgement for the input interaction (ok, ko). Whenever an instance of $DIIR_{\mathcal{C}}$ receives a message to be sent to an instance of $\mathcal{C}$, it verifies whether the corresponding instance of the first SAET is activated by interacting with $DIM_{\mathcal{C}}$. If so, the instance of $DIIR_{\mathcal{C}}$ forwards the message to such an instance of $\mathcal{C}$ and sends back a positive acknowledgement, otherwise it sends back a negative acknowledgement. The full behavior of instance i of $DIIR_{\mathcal{C}}$ is shown below:

$DIIRBeh^i_{\mathcal{C}}(;\text{Id}\ sender,\text{Id}\ receiver,\text{int}\ probe,\underline{x}) \triangleq$
  $input^i?(sender, receiver, \underline{x}).$
    $send\_probe^i_{\mathcal{C}}!(receiver).get\_probe^i_{\mathcal{C}}?(probe).$
      if $(probe = 0)$ then
        $ko^i.continue^i_{\mathcal{C}}.DIIRBeh^i_{\mathcal{C}}$
      else
        if $(receiver = 1)$ then
          $forward\_input^i_1!(\underline{x}).ok^i.continue^i_{\mathcal{C}}.DIIRBeh^i_{\mathcal{C}}$
        else if $(receiver = 2)$ then
          $forward\_input^i_2!(\underline{x}).ok^i.continue^i_{\mathcal{C}}.DIIRBeh^i_{\mathcal{C}}$
        else if ...
          ...
        else
          $forward\_input^i_n!(\underline{x}).ok^i.continue^i_{\mathcal{C}}.DIIRBeh^i_{\mathcal{C}}$

Note that a message is not forwarded to its destination instance of $\mathcal{C}$ both in the case in which the instance of $\mathcal{C}$ does not exist and in the case in which the instance of $\mathcal{C}$ is not ready to accept it. Whenever an AEI contains an output interaction attached to the prototype instance of $\mathcal{C}$, every occurrence of such an interaction is transparently modified in order to manage the ok/ko response. The output interaction must be declared as being persistent or nonpersistent. In the persistent case, the output interaction is considered successful if the destination instance of $\mathcal{C}$ exists and is willing to accept that interaction; should the destination instance of $\mathcal{C}$ be not existing, the output interaction is frozen and the AEI issuing it is kept waiting for the destination instance of $\mathcal{C}$ to come into existence. In the nonpersistent case, instead, the output interaction is considered successful even if the destination instance of $\mathcal{C}$ does not exist, in which case the AEI issuing the output interaction goes ahead with its computation. Every subterm of the form $output!(self, id, \underline{e}).E$ of the considered AEI must be replaced with $A_{output,E}(self, id, \underline{e})$ where

$A_{output,E}(\text{Id}\ self, \text{Id}\ id, \underline{x}) \triangleq$
  $output!(self, id, \underline{e}).(ok.E + ko.A_{output,E}(self, id, \underline{e}))$

if $output$ is persistent and

$A_{output,E}(\text{Id}\ self, \text{Id}\ id, \underline{x}) \triangleq$
  $output!(self, id, \underline{e}).(ok.E + ko.E)$

if $output$ is nonpersistent. Note that, if the destination instance of $\mathcal{C}$ exists, $A_{output,E}(self, id, \underline{e})$ gets blocked both in the persistent and in the nonpersistent case until the destination instance of $\mathcal{C}$ is ready to accept the output interaction.

The fourth SAET is a dynamic output instance router called $DOIR_{\mathcal{C}}$, which handles the attachments shared by the output interactions of the instances of $\mathcal{C}$. $DOIR_{\mathcal{C}}$ has as many instances as there are output interactions, where each instance is equipped with as many input interactions as there are instances of $\mathcal{C}$ that can send out a message through the considered output interaction, an output interaction with the outside to suitably forward the messages to be sent out (*forward_output*), two input interactions for receiving the positive/negative acknowledgement for the output interaction (*ok*, *ko*), and two sets of interactions for forwarding the positive/negative acknowledgement for the output interaction (*forward_ok*, *forward_ko*). Whenever an instance of $DOIR_{\mathcal{C}}$ receives a message to be sent out from an instance of $\mathcal{C}$, it forwards the message to the receiver. If the receiver is a DAEI, $DOIR_{\mathcal{C}}$ waits for an acknowledgement as the corresponding SAEI may not be activated at that instant, which is then forwarded back to the instance of $\mathcal{C}$ issuing the message if it is still activated. The full behavior of instance i of $DOIR_{\mathcal{C}}$ is shown below; for the sake of simplicity, we show the behavior only w.r.t. instance j of the first SAET:

$DOIRBeh^i_{\mathcal{C}}(;\mathrm{Id}\ sender, \mathrm{Id}\ receiver, \mathrm{int}\ probe, \underline{x}) \triangleq$
$output^i_j?(sender, receiver, \underline{x}).$
$forward\_output^i!(sender, receiver, \underline{x}).$
$(ok^i.send\_probe^i_{\mathcal{C}}!(sender).get\_probe^i_{\mathcal{C}}?(probe).$
if $(probe = 0)$ then
$DOIRBeh^i_{\mathcal{C}}$
else
$forward\_ok^i_j.DOIRBeh^i_{\mathcal{C}} +$
$ko^i.send\_probe^i_{\mathcal{C}}!(sender).get\_probe^i_{\mathcal{C}}?(probe).$
if $(probe = 0)$ then
$DOIRBeh^i_{\mathcal{C}}$
else
$forward\_ko^i_j.DOIRBeh^i_{\mathcal{C}})$

4. THE BULLY ALGORITHM

In this section we show how to model with ΔPADL the dynamic framework in which a leader election algorithm, called the bully algorithm [4], operates in case of coordinator failure or destruction. Assume that every process has a unique number that identifies it. When a process P notices that the coordinator is no longer responding to requests, it initiates an election by sending an election message to all the processes with higher numbers. If no one responds, P wins the election and becomes the coordinator. If one of the higher-ups answers, it takes over and P job is done. At any moment, a process can get an election message from one of its lower-numbered colleagues. When such a message arrives, the receiver sends an ok message back to the sender. The receiver then holds an election, unless it is already holding one. Eventually, all the processes give up but one, and that one is the new coordinator. It announces its victory by sending all the lower-numbered processes a message telling them that it is the new coordinator. If a process that was previously down comes back up or a new process is spawned that joins the group, it holds an election. If it happens to be the highest-numbered process currently running, it will win the election and become the coordinator.

The ΔPADL description of the bully algorithm is hierarchical. The primary architectural type, *BullyAlgorithm* with parameters *max_proc_id* and *timeout_p*, defines the dynamic framework in which the processes operate. It is based on two prototype instances of a DAET representing a single process, whose behavior is given by an invocation of the secondary architectural type, and an instance of a SAET for creating/destroying the DAEIs at run time. Throughout the computation, the topology may evolve until *max_proc_id* DAEIs are simultanelously active:

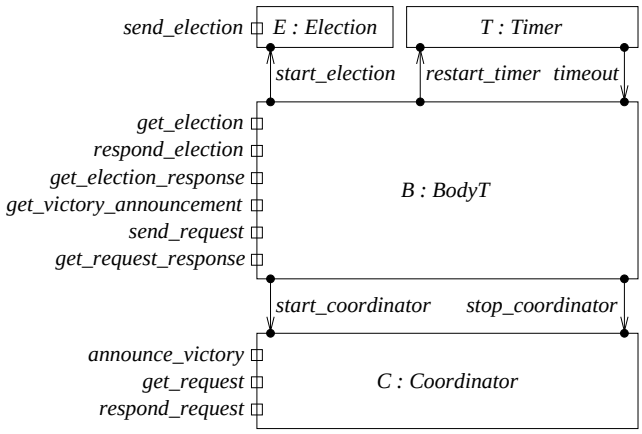

Figure 3: Flow graph of a single process

archi_type *BullyAlgorithm*(**Id** *max_proc_id*, **int** *timeout_p*)
archi_elem_types
dynamic_elem_type *BullyT*(**Id** *max_proc_id*)
behavior *Bully* $\triangleq$ *Process*(;; *max_proc_id*, *timeout_p*)
interactions
input
get_election, *get_election_response*,
get_victory_announcement,
get_request, *get_request_response*
output nonpersistent
send_election, *respond_election*, *announce_victory*,
send_request, *respond_request*
max_inst_no *max_proc_id*
static_elem_type *CreatorT*(**Id** *max_proc_id*)
behavior
$Creator(;\mathbf{Id}\ id) \triangleq$
$create_{BullyT}!(self).get\_id_{BullyT}!(receiver, id).Creator +$
$destroy_{BullyT}!(random(1..max\_proc\_id)).Creator +$
$\tau.Creator$
interactions
archi_topology
archi_elem_instances
B1, *B2* : *BullyT*(*max_proc_id*, *timeout_p*)
C : *CreatorT*(*max_proc_id*)
archi_interactions
archi_attachments
from *B1*/*B2.send_election* **to** *B2*/*B1.get_election*
from *B1*/*B2.respond_election*
to *B2*/*B1.get_election_response*
from *B1*/*B2.announce_victory*
to *B2*/*B1.get_victory_announcement*
from *B1*/*B2.send_request* **to** *B2*/*B1.get_request*
from *B1*/*B2.respond_request*
to *B2*/*B1.get_request_response*
end

The absence of explicit actual AETs and actual names for the architectural interactions in the invocation of the architectural type *Process* is a shorthand to indicate that the

actual AETs and the actual names for the architectural interactions coincide with the corresponding formal ones provided below in the definition of *Process*.

The secondary architectural type, *Process* with parameters *max_proc_id* and *timeout_p*, defines the behavior of a generic process based on four SAETs as shown in Fig. 3. The computation of the process is driven by the AET *BodyT*. It initially starts an election, then waits for some response. If no response is received from the higher-numbered processes before the timeout period expires, it becomes the coordinator, otherwise it waits for the victory announcement from one of the higher-numbered processes. If this announcement is received before the timeout period expires, then the process interacts with the new coordinator, otherwise it starts a new election. A new election is started also in the case in which the coordinator does not respond within the timeout period. Every late election, election response, and request response related message is simply ignored:

static_elem_type *BodyT*(**Id** *max_proc_id*, **int** *timeout_p*)
behavior
Body $\triangleq$ *start_election*!(*max_proc_id*).
restart_timer!(*timeout_p*).*Body′*
Body′ $\triangleq$ *get_election_response*.
restart_timer!(*timeout_p*).*Body1* +
timeout.*start_coordinator*.*Body2* +
get_election.*respond_election*!(*sender*).*Body′* +
get_victory_announcement.*Body1′*(*sender*) +
get_request_response.*Body′*
Body1 $\triangleq$
get_victory_announcement.*Body1′*(*sender*) +
timeout.*Body* +
get_election.*respond_election*!(*sender*).*Body1* +
get_election_response.*Body1* +
get_request_response.*Body1*
Body1′(**Id** *coord*;) $\triangleq$ *send_request*!(*coord*).
restart_timer!(*timeout_p*).*Body1″*(*coord*) +
get_election.*respond_election*!(*sender*).*Body* +
get_victory_announcement.*Body1′*(*sender*) +
get_election_response.*Body1′*(*coord*) +
get_request_response.*Body1′*(*coord*)
Body1″(**Id** *coord*;) $\triangleq$
get_request_response.*Body1′*(*coord*) +
timeout.*Body* +
get_election.*respond_election*!(*sender*).*Body* +
get_victory_announcement.*Body1′*(*sender*) +
get_election_response.*Body1″*(*coord*)
Body2 $\triangleq$ *get_election*.*respond_election*!(*sender*).
stop_coordinator.*Body* +
get_victory_announcement.*stop_coordinator*.
Body1′(*sender*) +
get_election_response.*Body2* +
get_request_response.*Body2*
interactions
input *get_election*, *get_election_response*,
get_victory_announcement,
get_request_response, *timeout*
output *start_election*, *restart_timer*,
start_coordinator, *stop_coordinator*
output nonpersistent *respond_election*, *send_request*

The AET *ElectionT* starts an election by sending a message to all the higher-numbered processes:

static_elem_type *ElectionT*
behavior
Election(; **Id** *higher_proc*) $\triangleq$
start_election?(*higher_proc*).*Election′*(*higher_proc*)
Election′(**Id** *higher_proc*;) $\triangleq$
if (*higher_proc* > *self*) **then**
send_election!(*higher_proc*).*Election′*(*higher_proc* − 1)
else
Election
interactions
input *start_election*
output nonpersistent *send_election*

The AET *TimerT* is a clock employed to signal the timeout expiration in case of election started by the process or request sent by the process to the coordinator:

static_elem_type *TimerT*
behavior
Timer(; **int** *remaining_time*) $\triangleq$
restart_timer?(*remaining_time*).*Timer′*(*remaining_time*)
Timer′(**int** *remaining_time*; **int** *new_remaining_time*) $\triangleq$
if (*remaining_time* = 0) **then**
timeout.*Timer* +
restart_timer?(*new_remaining_time*).
Timer′(*new_remaining_time*)
else
tick.*Timer′*(*remaining_time* − 1) +
restart_timer?(*new_remaining_time*).
Timer′(*new_remaining_time*)
interactions
input *restart_timer*
output *timeout*

The AET *CoordinatorT* represents the behavior of the process when it is the coordinator. According to the bully algorithm, then newly elected coordinator communicates its victory to all the lower-numbered processes, then responds to the incoming requests:

static_elem_type *CoordinatorT*
behavior
Coordinator $\triangleq$
start_coordinator.*Coordinator′*(*self* − 1)
Coordinator′(**Id** *lower_proc*;) $\triangleq$
if (*lower_proc* > 0) **then**
announce_victory!(*lower_proc*).
Coordinator′(*lower_proc* − 1)
else
Coordinator″
Coordinator″ $\triangleq$
get_request.*respond_request*!(*sender*).*Coordinator″* +
stop_coordinator.*Coordinator*
interactions
input *start_coordinator*, *stop_coordinator*, *get_request*
output nonpersistent *announce_victory*, *respond_request*

As can be observed, the architectural description of the bully algorithm is fully parameterized w.r.t. the number of processes, which are freely created/destroyed at run time. We only needed to specify the behavior of a single generic process and declare the related connections through a pair of prototype instances. On the analysis standpoint, the same

techniques developed for PADL can be reused. As an example, it can easily be verified that E, T, and C in Fig. 3 are compatible with B, which is deadlock free. As a consequence, we can conclude that a single process behaving according to the bully algorithm is deadlock free. For a deeper investigation of the properties of the bully algorithm, like global deadlock freedom and termination, it is worth recalling that our framework supports standard model checking techniques.

5. CONCLUSION

In this paper we have presented an enhancement of PADL called ΔPADL, which supports the description of software architectures where some components can be dynamically created/destroyed throughout the execution. The novelty of our approach in the field of formal method based ADLs is that every ΔPADL specification is parameterized w.r.t. the maximum number of dynamic components of each type and simply requires the provision of the connections for a prototype dynamic component of each type. The strength of ΔPADL is that its specifications, besides scaling w.r.t. the maximum number of dynamic components of each type, make the component creation/destruction related details transparent to the designer, thanks to suitable instance managers and routers, and can undergo to the same architectural checks as PADL specifications.

Our approach to the description and analysis of dynamic software architectures is closely related to the approach of Dynamic Wright [1], due to its process algebraic nature and its static representation of all the possible system configurations. In Dynamic Wright a separation of concerns is realized, which requires the designer to explicitly provide a configuror that defines the initial topology of the system as well as the reconfiguration actions (in terms of creating/destroying components and attaching/detaching connections) to be undertaken whenever some triggering events happen. The semantics is then given by translation into CSP [5] after suitably tagging every action in the behavior of the components, so that all the possible system configurations are statically represented and the architectural checks for Wright [2] can be reused with some minor modifications. A drawback of the approach of Dynamic Wright is that the provision of the description of a configuror accounting for all the possible system reconfigurations may not be an easy task, especially in the case of very complex systems. Additionally, the resulting specifications are not always scalable w.r.t. the dynamic components and may require the introduction of global knowledge about the existence of dynamic components into the behavior of other components. As an example, if we try to model the bully algorithm with Dynamic Wright, we have that it is not possible to specify a single process component nor a single process connector because the definition of some of its ports/roles depends on the id of the process. E.g. if the maximum number of processes is 10, then process 7 can receive an election message through its connector from one of the processes 1 to 6, while process 4 can receive an election message through its connector only from one of the processes 1 to 3.

The view taken in ΔPADL is more code oriented, because the actions changing the system structure (create/destroy) are distributed among the components in the places where they are needed. This brings the advantage that in ΔPADL the architect is not obliged to provide the detailed description of an external configuror. The way in which the components are dynamically created/destroyed in ΔPADL is similar to that of Rapide [6], where an object oriented view is essentially taken. In Rapide a poset execution model is considered, so that architectural properties are studied on sets of execution traces via simulation rather relying on local static checks like in ΔPADL. Finally, dynamic software architectures can be described with Darwin [7]. Unlike ΔPADL, Darwin is a purely declarative language concerned with structural aspects only, hence architectural analysis techniques considering the system component behavior do not apply.

As far as future work is concerned, on the modeling side we would like to add the capability of dynamically varying the attachments among the AEIs, in order to achieve a full dynamicity of the system topology. This could be realized through a transparent dynamic attachment manager that, similarly to the dynamic instance manager, maintains a table of the activated/deactivated attachments. On the analysis side, we would like to develop more efficient architectural checks for ΔPADL. Here the observation is that the behavior of a ΔPADL specification can be seen as being bidimensional: one dimension is related to a configuration change due to the creation/destruction of a DAEI, the other dimension is related to a state change within a configuration due to an action performed by an AEI or several synchronized AEIs. Given this view, it is desirable to apply architectural checks to single configurations in order to derive information about the whole system.

6. REFERENCES

[1] R. Allen, R. Douence, D. Garlan, *"Specifying and Analyzing Dynamic Software Architectures"*, in Proc. of FASE 1998, LNCS 1382, Lisbon (Portugal), 1998

[2] R. Allen, D. Garlan, *"A Formal Basis for Architectural Connection"*, in ACM TOSEM 6:213-249, 1997

[3] M. Bernardo, P. Ciancarini, L. Donatiello, *"Architecting Software Systems with Process Algebras"*, Tech. Rep. UBLCS-2001-07, University of Bologna (Italy), 2001

[4] H. Garcia-Molina, *"Elections in a Distributed Computing System"*, in IEEE TOC 31:48-59, 1982

[5] C.A.R. Hoare, *"Communicating Sequential Processes"*, Prentice Hall, 1985

[6] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, W. Mann, *"Specification and Analysis of System Architecture Using Rapide"*, in IEEE TSE 21:336-355, 1995

[7] J. Magee, N. Dulay, S. Eisenbach, J. Kramer, *"Specifying Distributed Software Architectures"*, in Proc. of (ESEC 1995), LNCS 989:137-153, Barcelona (Spain), 1995

[8] R. Milner, *"Communication and Concurrency"*, Prentice Hall, 1989

[9] D.E. Perry, A.L. Wolf, *"Foundations for the Study of Software Architecture"*, in ACM SIGSOFT Software Engineering Notes 17:40-52, 1992

[10] M. Shaw, D. Garlan, *"Software Architecture: Perspectives on an Emerging Discipline"*, Prentice Hall, 1996