ACME vs PDDL: support for dynamic reconfiguration of software architectures

Jean-Eudes Méhus, Thais Batista, Jérémy Buisson

ACME vs PDDL: support for dynamic reconfiguration of software architectures

Jean-Eudes Méhus
Écoles de St-Cyr Coëtquidan
Guer, France
jean-eudes.mehus@st-cyr.terre-net.defense.gouv.fr

Thais Batista
Federal University of Rio Grande do Norte
Natal (RN) Brazil
thais@dimap.ufrn.br

Jérémy Buisson
UEB / Écoles de St-Cyr Coëtquidan / Université de Bretagne Sud
Guer, France
jeremy.buisson@st-cyr.terre-net.defense.gouv.fr

ABSTRACT

On the one hand, ACME is a language designed in the late 90s as an interchange format for software architectures. The need for reconfiguration at runtime has led to extend the language with specific support in Plastik. On the other hand, PDDL is a predicative language for the description of planning problems. It has been designed in the AI community for the International Planning Competition of the ICAPS conferences. Several related works have already proposed to encode software architectures into PDDL. Existing planning algorithms can then be used in order to generate automatically a plan that updates an architecture to another one, i.e., the program of a reconfiguration. In this paper, we improve the encoding in PDDL. Noticeably we propose how to encode ADL types and constraints in the PDDL representation. That way, we can statically check our design and express PDDL constraints in order to ensure that the generated plan never goes through any bad or inconsistent architecture, not even temporarily.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering

Keywords

dynamic reconfiguration, ACME, PDDL, planning, software architecture

1. INTRODUCTION

With long-running software systems, we sometimes want to change the software program at runtime. That way, one can deploy new features or fix bugs without any service disruption. Continuous operation and critical systems have such requirements for dynamic reconfiguration. Dynamic reconfiguration have been studied at different levels (control flow [12], functions [3, 24], objects [2, 11], components [6, 7, 10]). In this paper we consider only structural changes of the component-based software architecture. At this level, dynamic reconfigurations typically consist in adding and removing components and connectors, as well as changing the connections between architectural elements.

Several software architecture reconfiguration languages have been proposed [6, 10, 27] to let developers program reconfigurations by-hand. These languages provide primitive operations that add, remove and modify architectural elements at runtime. The developer uses these operations in combination with usual connectives (sequence, iteration, condition) to describe imperatively how to transform the software architecture to the new desired version.

In order to relieve from the burden of programming reconfigurations, some related works have tried to automate the issuing of reconfiguration instructions [1, 4, 5, 13, 20, 23]. These techniques compare the original and target architectures in order to identify the changes. Then a tool computes a suitable sequence of primitive operations that performs the reconfiguration.

Instead of designing domain-specific algorithms, it is appealing to reuse off-the-shelf techniques to generate reconfiguration scripts. That way we can benefit from advances and expertise in other research fields. Automatic action planning from the AI community is one candidate technology. It is a field of research that focuses on the generation of a sequence of actions that brings a system from an initial state to a goal state, based on the specification of all possible actions. Planners such as POPF [9], Fast Downward [18], LAMA [25] and Madagascar [26] are general-purpose tools that can be used if software architecture reconfigurations can be translated to planning problems.

In the AI community, the *de facto* standard description language for planning problems is PDDL [16]. This language is designed and used by the International Planning Competition of the ICAPS conferences. PDDL is therefore a widely spread language that is implemented by many planners.

As noticed by André *et al.* [1], PDDL let us switch easily from one planner to another. Furthermore, regular planners usually generate better reconfiguration scripts than simple

domain-specific heuristics such as [23]. However, in the current state of the art, constraints (coming from either the software architecture, the component model or the execution platform) are not taken into account. Even if the specification of reconfiguration operations is correctly designed, there is no guaranty (in the current state of the art) that the planner cannot generate operations that infringe any constraint. In order to address this issue, we propose in this paper that we statically verify general constraints and that we embed the other constraints in the planning problem.

This paper improves the current state of the art (described in Section 2) in order to take into account constraints. Our presentation is based on a synthetic client-server example described in Section 3. We program this example using the ACME [14] architecture description language in Section 4. As exposed, ACME is richer than the architecture description languages used by Arshad *et al.*, André *et al.* and Ingstrup *et al.* Indeed, ACME supports the component-and-connector paradigm with types, as well as invariants and architectural styles. By means of its extension Armani [21], ACME let the software architect state constraints. ACME has also built-in support for dynamic reconfiguration thanks to Plastik [6]. In this paper, we do not refrain from defining new reconfiguration operations, e.g., modifying types. Section 5 describes the same example using PDDL. We first presents this language. We give the specification of the primitive reconfiguration operations as well as the predicates that we use to encode software architectures. A set of invariants relates the predicates according to the ACME component model. In Section 6 we show how some of the constraints coming either from the ACME language itself (e.g., typing) or from architectural styles (e.g., client-server) can be checked statically, and therefore we show that the planning problem is consistent with these invariants. The remaining invariants can be embedded in the planning problem in order to tell the planner not to infringe any of the constraints. Section 7 contains a discussion of our results, reports our first experiments and concludes the paper with future works.

2. STATE OF THE ART

In the current state of the art, regular PDDL planners from the AI community have already been used successfully to generate automatically reconfiguration scripts [1, 4, 5, 13, 20]. With these systems, the original architecture and the target one are encoded together as a planning problem. Each reconfiguration primitive is modelled as an action in a planning domain. From these descriptions, a planner generates a sequence (possibly partially ordered if concurrency is supported) of primitive operations that brings the architecture from its original state to the target configuration. It is important that the planning problem states clearly the constraints imposed by, e.g., the execution platform in order that the planner does not generate an infeasible plan.

We build on the previous state-of-the-art results of Arshad *et al.* [4, 5], André *et al.* [1], Ingstrup *et al.* [20] and El Maghraoui *et al.* [13] in order to improve their techniques. All of these previous works translate reconfigurations to PDDL planning problems. They define a set of predicates to describe a software architecture as the conjunction of logical facts. For example in [5], the `connected−component` predicate states whether a given component and a given connector are connected to one another. A set of actions models the semantics (preconditions and effects) of the reconfiguration primitive operations. An action named `connect−component` for instance requires that a component and a connector are instantiated and that none of them is connected; its effect establishes the `connected−component` fact. André *et al.*, Ingstrup *et al.* and El Maghraoui *et al.* [13] do the same even if they define different sets of predicates and actions, taking into account their respective contexts (i.e., their component model and their execution platform).

El Maghraoui *et al.* [13] propose in addition an encoding for properties that is based on predicates. A predicate named `set` states whether a given property of a given object has a value. For each property of each type, a specific predicates relates an object to the value of the attribute.

In summary, the main achievements and limits of previous works are:

- Arshad *et al.* [4, 5] use the approach in the context of a component-and-connector ADL. However, their model does not take into account ports, roles, types, constraints or architectural styles.
- André *et al.* [1] use the approach in the context of DiVA ART [22]. They support component instances and types as well as ports. However, they do not have the connector concept and they do not support constraints or architectural styles.
- Ingstrup *et al.* [20] have done similar experiments on the generation of deployment plans for OSGi bundles. However, they do not consider types or architectural styles.
- El Maghraoui *et al.* [13] use PDDL planning in order to generate deployment plans for datacenters managed by tools such as IBM Tivoli Provisioning Manager [19]. They support object-relationship models with properties. While types are taken into account, their structure is not modelled in PDDL. They do not support constraints.

Each of these previous works assume consistency rules that relate the predicates. For instance, El Maghraoui *et al.* [13] assume the consistency rules that states that if a property is set for an object then the property has a value for that object; at there is at most one value per property-and-object pair; and so on. Only few of such rules are explicitly given. While Ingstrup *et al.* have analyzed their reconfiguration actions using Alloy, they acknowledge that the version used with AI planning is different from the verified one [17]. None of Arshad *et al.* [4, 5], André *et al.* [1] or El Maghraoui *et al.* [13] take any preventive measure in regard to such constraints. None of these works avoid that the planner generates inconsistent, ill-typed or non-conformant architectures as intermediate reconfiguration steps. This is the point we address in this paper.

3. A RUNNING EXAMPLE

Figure 1 depicts the architecture that we use as the running example of this paper. It is a client-server architecture, which contains two components (`Client` is the client;

Table 1: ACME extensions for programmed reconfiguration.

Reconfiguration statements	Description
`On (<Armani exp>) do {<statements>}`	expresses runtime conditions under which programmed reconfigurations should take place, and a specification (in terms of the other reconfiguration statements) of what should change.
`Detach <element> from <element>`	removes an attachment between a port and a role.
`Remove <element>`	destroys an existing component, connector or representation.
`Dependencies {<ACME statements>}`	expresses runtime dependencies among components/connectors (e.g., if X is to be removed, Y should be removed also).

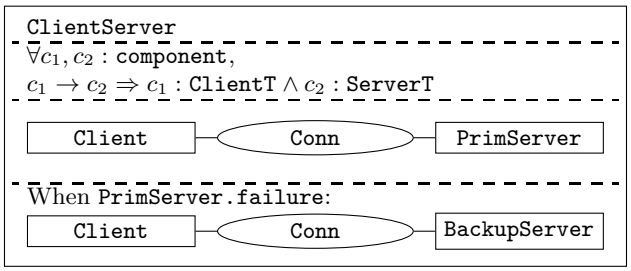

Figure 1: Architecture of the running example.

`PrimServer` is the server) and a connector named `Conn`. An invariant is attached to this architecture in order to ensure the client-server style. It states that for any pair c_1, c_2 of components, if c_1 is connected to c_2, then c_1 conforms to type `ClientT` and c_2 conforms to type `ServerT`. This invariant prevents from connecting one client component to another client, or one server component to another server.

Regarding reconfiguration, one may want to replace the primary server component `PrimServer` in case it fails. Obviously the reconfigured architecture, i.e., using a backup server named `BackupServer` in place of `PrimServer`, conforms to the invariant.

In PDDL, the states of a system, including the initial state and the goal state, are described by formulas in the first-order predicate logic. A planner may generate any plan that conforms to the goal. Conformance is defined by implication: as long as the final state implies the goal, the plan is acceptable. Consequently, the resulting architecture is not exactly the same as the goal architecture. For example, the resulting architecture may contain components, connectors and bindings that are not required, as long as they are not forbiden by the goal formula.

Therefore if we don't take any preventive measure, despite the target architecture respects invariants and constraints, the resulting architecture (after effective execution of the generated reconfiguration) may not.

A planner may execute any action as long as it can be executed. For instance, a planner may freely trigger an action, then undo that action at the next step. While planner implementors do their best not to issue useless actions, it is not mandatory that a planner generate optimal plans.

Therefore if we don't take any preventive measure, some intermediate steps shall be invalid with respect to invariants and constraints even if the resulting architecture is conformant.

In our example, we would like that the connector is detached from `PrimServer` before it is connected to the backup server; that the backup server is instantiated before its port is bound. The reconfiguration designer may also want to state that the architecture must continuously conform to its invariant during the reconfiguration; or a specific invariant shall be given for the time of the reconfiguration.

While the current state-of-the-art techniques [1, 4, 5, 13, 20] propose carefully designed planning problems, none of these previous works describe how to verify that they enforce the constraints. Despite they verify reconfiguration operations using Alloy [17], Ingstrup *et al.* acknowledge that the PDDL specification is not the verified one. In addition, in case some constraints cannot be checked statically, none of these previous works propose how to embed the constraints in the planning problem. None of [1, 4, 5, 13, 17, 20] take into account invariants given by the software architect.

4. THE EXAMPLE USING ACME ADL

ACME [14] is an extensible generic ADL that provides a syntax for representing structures and an annotation mechanism for describing additional semantics. The ACME core concepts are:

- Components: the basic building blocks in an ACME description of a system. Components expose their functionality through their ports. A port represents a point of contact between the component and its environment.
- Connectors: represent communication glue that captures the nature of an interaction between components. *Ports* are bound to ports on other components using connectors. Like components, connectors may be used to model a variety of different sorts of interactions under a number of different models. A connector includes a set of interfaces in the form of *roles*.
- Systems: describe a set of components and connectors, and how they interact. A property might also be used to represent properties of the environment in which the system is operating, or "global" properties that apply to all elements of the system. The graph of a system (how everything is connected) is defined by a set of *attachments*.
- Representations: are alternative decompositions of a given component; they reify the notion that a compo-

nent may have multiple alternative implementations. Elements in ACME may have more than one representation.

- Element types: are defined in the same way as instances; they define a prototype that is instantiated by copying its structure. The ACME type model states that all instances of a type must include the structure defined by this type. For properties, this means that if a set of properties is defined for a particular type, any instances must have the same properties.
- Attachments: define a set of port/role associations.
- Properties: are <name, type, value> triples that annotate to any of the above ACME elements. ACME allows user-defined property types that may be defined in terms of built-in property data types. Systems, components, connectors, ports and roles may include a list of properties and a list of representations.
- Architectural styles: define sets of types of components, connectors, properties, and sets of rules that specify how elements of those types may be legally composed in a reusable architectural domain.

Armani [21] extends ACME with a language based on first-order predicates used to express architectural constraints over architectures. For example, it can be used to express constraints on system composition, behaviour, and properties. Constraints are defined in terms of so-called invariants which in turn are composed of standard logical connectives and predicates (both built-in and user-defined) which are referred to as functions.

ACME may also be used as a way of representing reconfigurable architectures by expressing the possible reconfigurations in terms of the ACME structures. For example, a system might include properties that describe components that may be added at run-time and how to attach them to the current system. This means that ACME does not include first class elements to describe dynamic reconfiguration of the architecture. In other words, dynamic reconfiguration is not originally addressed by ACME but it can be handled using the extensible mechanism of the language. In order to address this issue, Plastik [6] defines ACME extensions to represent different types of reconfigurations at the architecture level. Table 1 summarizes the set of such ACME extensions for programmed reconfiguration, that is, reconfiguration that can be foreseen at design time. Ad-hoc reconfiguration, which involves changes unforeseen at design time, can be specified at the architecture level by submitting a partial architecture specification to a configurator. At the architecture level Plastik is based on styles to constrain the allowable range of permissible ad-hoc reconfigurations.

Figure 2 illustrates the use of Armani and Plastik in our example. A programmed reconfiguration specifies the removal of the `PrimServer` component if it fails: the `Conn` connector and the `PrimServer` component are disconnected, then `PrimServer` is removed, and a `BackupServer` component is inserted and attached to `Conn`.

The description of an architecture in ACME with dynamic reconfiguration statements follows the following steps:

```
Family ClientServerFam extends PlastikMF with {
 Component Type ClientT = { Port request = new RequiredPort; }
 Component Type ServerT = { Port service = new ProvidedPort; }
 Invariant Forall c1, c2: Component in self.Components |
  connected(c1, c2) -> (satisfiesType(c1, ?ClientT?) and
                        satisfiesType(c2, ?ServerT?));
}
System ClientServer = new ClientServerFam extended with {
 Component Client = new ClientT;
 Component PrimServer = new ServerT extended with {
  Property failure: boolean = false;
 }
 Connector Conn = { Role requestor; Role servicer; }
 Attachments { Client.request to Conn.requestor;
               Conn.servicer to PrimServer.service; }
 On (PrimServer.failure == true) do {
  Detach Conn.servicer from PrimServer.service;
  Remove PrimServer;
  Component BackupServer = new ServerT extended with {
   Dependencies {
    Attachments { Conn.servicer to BackupServer.service; }
  }
 }
}
```

Figure 2: A dependable client-server in Plastik.

- Identify the concepts in the source model that correspond to the ACME architectural concept: system, component, connector, port, role or representation
- Define a family or set of families for the model. Define a component, connector, port or role type to represent each of the architectural concepts
- Define a set of property types, which will make up a property language for describing elements in the model
- Define the reconfiguration actions

5. THE EXAMPLE USING PDDL

PDDL is a standard encoding language for planning tasks. The components of a PDDL planning task are: (i) *Objects* (ii) *Predicates*: properties of objects that can be true or false, as in first order logic; (iii) *Initial state*: the list of all facts that are true in the initial state; (iv) *Goal specification*: the objective of the problem that specifies what need to be true at the end of the plan; (v) *State trajectory constraints*: a logical formula used to restrict the space of states; (vi) *Actions/Operators*: ways of changing the truth and falsity of facts. Actions are parameterized with objects. An action is composed of a *precondition*, that states the constraints to the action be executed, and an *effect* that lists the facts that become true or false after the execution of the action.

The planning task is usually split into two files:

1. A domain file for the definition of the domain predicates and actions. To some extent, the domain defines the language used to describe situations and planning problems in a specific application.
2. A problem file containing the objects of the problem instance, initial state and goal specifications.

In summary, the description of predicates structures the representation of states; the description of actions characterizes

```
(define (domain <domain name>)
   <PDDL code for predicates>
   <PDDL code for first action>
   [...]
   <PDDL code for last action>
)
```

Figure 3: Domain File in PDDL.

```
(define (problem <problem name>)
   (:domain <domain name>)
   <PDDL code for objects>
   <PDDL code for initial state>
   <PDDL code for goal specification>
)
```

Figure 4: Problem File in PDDL.

domain behaviours. Predicates and actions (the domain) are separated from the description of specific instance objects, initial conditions, and goals that characterize a problem instance. A planning problem is created by joining a domain description with a problem description. The same domain description can be joined with many different problem descriptions to yield different planning problems in the same domain. The structure of a domain file is depicted in Figure 3. The structure of a problem file is depicted in Figure 4.

In this work, we inspire from previous works: most architecture elements (components, connectors, ports, roles, systems and types) are encoded as objects. The relationships between elements (including attachments) are encoded as facts, thanks to predicates. Each reconfiguration primitive operation reflects as a PDDL action. Architectural styles are either checked statically or programmed as trajectory constraints. In this paper, we ignore representations and properties. We do not consider the reconfiguration of invariants or styles.

5.1 A domain for software architectures

Like [5], we use PDDL types only to classify objects depending on the kind of architecture elements. The reason why we don't encode ACME types using PDDL types is because the two type systems are different. On the one side, ACME component and connector types are structural types: a component is an instance of a type if and only if it contains at least the same structure as the one described in the type. On the other side, PDDL types are nominal types. The type of an object is given by name. Two types with different names are different types.

The relationship between components and connectors instances on the one side, and types on the other side is modelled by `has−component−type` and `has−connector−type`.

The containment relationships are implemented by a set of predicates, one per level in the hierarchy. Table 2 summarizes these predicates. Our model assumes that if a component `c` has type `t`, then `c` contains the same ports as `t`. The same invariant applies to connectors and roles.

The `bound` predicate binds a port of a component to a role of a connector. The predicate named `exist−component` (resp. `exist−connector`) states that a component (resp. connector) is instantiated.

We also define negative predicates: the predicate named `unbound−port` (resp. `unbound−role`) states that a port of a component (resp. a role of a connector) is not bound to any role (resp. port).

We model each reconfiguration primitive operation as an action, expliciting the preconditions and effects on the architecture in terms of the above predicates.

- `create−component`: instantiate a component named `?c` in a system `?s`.
  Precondition: ¬`exist−component` (`?c`).
  Effects: `exist−component` (`?c`),
  `contains−component` (`?s`, `?c`).

- `create−connector`: instantiate a connector named `?c` in a system `?s`.
  Precondition: ¬`exist−connector` (`?c`).
  Effects: `exist−connector` (`?c`),
  `contains−connector` (`?s`, `?c`).

- `remove−component`: remove a component named `?c` from a system `?s`.
  Precondition: `contains−component` (`?s`, `?c`).
  Effects: ¬`exist−component` (`?c`),
  ¬`contains−component` (`?s`, `?c`).

- `remove−connector`: remove a connector `?c` from a system `?s`.
  Precondition: `contains−connector` (`?s`, `?c`).
  Effects: ¬`exist−connector` (`?c`),
  ¬`contains−connector` (`?s`, `?c`).

- `attach`: bind a port `?p` of a component `?c` to a role `?r` of a connector `?co`.
  Precondition: `exist−component` (`?c`),
  `exist−connector` (`?co`), `has−port` (`?c`, `?p`),
  `has−role` (`?co`, `?r`), `unbound−port` (`?c`, `?p`),
  `unbound−role` (`?co`, `?r`).
  Effects: ¬`unbound−port` (`?c`, `?p`),
  ¬`unbound−role` (`?co`, `?r`), `bound` (`?c`, `?p`, `?co`, `?r`).

- `detach`: unbind a port `?p` of a component `?c` from a role `?r` of a connector `?co`.
  Precondition: `bound` (`?c`, `?p`, `?co`, `?r`).
  Effects: ¬`bound` (`?c`, `?p`, `?co`, `?r`),
  `unbound−port` (`?c`, `?p`), `unbound−role` (`?co`, `?r`).

As we model the types of architectural elements, we can also define operations that affect types. As in ACME each element has its own type (defined by its own structure) we can define operations that consistently change a component or a connector and its type at once. No question arises whether such modifications should propagate to a whole group of instances as each type is bound to one instance at most.

- `add−port`: add a port `?p` to a component `?c` of type `?t`.
  Precondition: `has−component−type` (`?c`, `?t`),

Table 2: Predicates for containment relationships.

	Types		Instances	
system			constains−component	contains−connector
component/connector	type−has−port	type−has−role	has−port	has−role

¬has−port (?c, ?p).
Effects: has−port (?c, ?p), type−has−port (?t, ?p), unbound−port (?c, ?p).

- add−role: add a role ?r to a connector ?c of type ?t.
  Precondition: has−connector−type (?c, ?t), ¬has−role (?c, ?r).
  Effects: has−role (?c, ?r), type−has−role (?t, ?r), unbound−role (?c, ?r).
- remove−port: remove a port ?p from a component ?c of type ?t.
  Precondition: has−component−type (?c, ?t), has−port (?c, ?p), unbound−port (?c, ?p).
  Effects: ¬has−port (?c, ?p), ¬type−has−port (?t, ?p).
- remove−role: remove a role ?r from a connector ?c of type ?t.
  Precondition: has−connector−type (?c, ?t), has−role (?c, ?r), unbound−role (?c, ?r).
  Effects: ¬has−role (?c, ?r), ¬type−has−role (?t, ?r).

In case a type is not bound to any instance, we can also reconfigure that type independently of any instance.

- add−type−port: add a port ?p to a component type ?t.
  Precondition: ∀?c : component, ¬has−component−type (?c, ?t), ¬type−has−port (?t, ?p).
  Effects: type−has−port (?t, ?p).
- add−type−role: add a role ?r to a connector type ?t.
  Precondition: ∀?c : connector, ¬has−connector−type (?c, ?t), ¬type−has−role (?t, ?r).
  Effects: type−has−role (?t, ?r).
- remove−type−port: remove a port ?p from a component type ?t.
  Precondition: ∀?c : component, ¬has−component−type (?c, ?t), type−has−port (?t, ?p).
  Effects: ¬type−has−port (?t, ?p).
- remove−type−role: remove a role ?r from a connector type ?t.
  Precondition: ∀?c : connector, ¬has−connector−type (?c, ?t), type−has−role (?t, ?r).
  Effects: ¬type−has−role (?t, ?r).

It is the role of the constraints such as subtyping or type satisfaction to restrict the use of the operation that affect types.

In this design, we assume that any situation conforms to the following consistency constraints. The equality = denotes the identity over PDDL objects (including components, connectors, their types, ...).

- Each component has a type:

  $\forall c : \texttt{component}, \exists t : \texttt{component-type}, \texttt{has-component-type}\,(c, t)$

- The type of a component is unique:

  $\forall c : \texttt{component}, \forall t_1, t_2 : \texttt{component-type}, \texttt{has-component-type}\,(c, t_1) \wedge \texttt{has-component-type}\,(c, t_2) \Rightarrow t_1 = t_2$

- Each component has a different type:

  $\forall c_1, c_1 : \texttt{component}, \forall t : \texttt{component-type}, \texttt{has-component-type}\,(c_1, t) \wedge \texttt{has-component-type}\,(c_2, t) \Rightarrow c_1 = c_2$

- A component and its type have the same ports:

  $\forall c : \texttt{component}, \forall t : \texttt{component-type}, \texttt{has-component-type}\,(c, t) \Rightarrow \forall p : \texttt{port}, \texttt{has-port}\,(c, p) \Leftrightarrow \texttt{type-has-port}\,(t, p)$

- A system contains only instantiated components:

  $\forall s : \texttt{system}, \forall c : \texttt{component}, \texttt{contains-component}\,(s, c) \Rightarrow \texttt{exist-component}\,(c)$

- Any instantiated component lies in a system:

  $\forall c : \texttt{component}, \texttt{exist-component}\,(c) \Rightarrow \exists s : \texttt{system}, \texttt{contains-component}\,(s, c)$

- A component is in a single system:

  $\forall c : \texttt{component}, \forall s_1, s_2 : \texttt{system}, \texttt{contains-component}\,(s_1, c) \wedge \texttt{contains-component}\,(s_2, c) \Rightarrow s_1 = s_2$

- Only instantiated components can be bound:

  $\forall c : \texttt{component}, \forall p : \texttt{port}, \forall co : \texttt{connector}, \forall r : \texttt{role}, \texttt{bound}\,(c, p, co, r) \Rightarrow \texttt{exist-component}\,(c)$

- Only the ports of a component can be bound:

  $\forall c : \texttt{component}, \forall p : \texttt{port}, \forall co : \texttt{connector}, \forall r : \texttt{role}, \texttt{bound}\,(c, p, co, r) \Rightarrow \texttt{has-port}\,(c, p)$

```
(:objects ClientServer            - system
          Client PrimServer       - component
          BackupServer            - component
          Conn                    - connector
          ClientT Client_type     - component_type
          ServerT PrimServer_type - component_type
          BackupServer_type       - component_type
          Conn_type               - connector_type
          request service         - port
          requestor servicer      - role)
```

Figure 5: The client-server PDDL objects.

```
(:init (exist_component Client)
       (exist_component PrimServer)
       (exist_connector Conn)
       (contains_component ClientServer Client)
       (contains_component ClientServer PrimServer)
       (contains_connector ClientServer Conn)
       (has_component_type Client Client_type)
       (has_component_type PrimServer PrimServer_type)
       (has_connector_type Conn Conn_type)
       (has_port Client request)
       (has_port PrimServer service)
       (has_role Conn requestor)
       (has_role Conn servicer)
       (type_has_port Client_type request)
       (type_has_port PrimServer_type service)
       (type_has_role Conn_type requestor)
       (type_has_role Conn_type servicer)
       (type_has_port ClientT request)
       (type_has_port PrimServer_type service)
       (bound Client request Conn requestor)
       (bound PrimServer service Conn servicer)
       (has_component_type BackupServer BackupServer_type))
```

Figure 6: The client-server architecture in PDDL.

- A port of a component can be bound only once:

$$\begin{array}{l}\forall c : \texttt{component}, \forall p : \texttt{port}, \\ \forall co_1, co_2 : \texttt{connector}, \forall r_1, r_2 : \texttt{role}, \\ \texttt{bound}\,(c, p, co_1, r_1) \wedge \texttt{bound}\,(c, p, co_2, r_2) \\ \Rightarrow co_1 = co_2 \wedge r_1 = r_2\end{array}$$

- A port is unbound iff it is not bound to any role:

$$\begin{array}{l}\forall c : \texttt{component}, \forall p : \texttt{port}, \\ \left(\begin{array}{l}\texttt{unbound}-\texttt{port}\,(c, p) \\ \Leftrightarrow \forall co : \texttt{connector}, \forall r : \texttt{role}, \\ \neg\texttt{bound}\,(c, p, co, r)\end{array}\right)\end{array}$$

The same constraints hold for connectors and roles.

5.2 The architecture in our PDDL domain

Figure 5 gives the PDDL listing for the objects in the architecture. It enumerates all of the architectural elements that are mapped onto PDDL objects: systems, components, connectors, ports, roles, component types and connector types. It contains all the objects that could exist before, during and after the reconfiguration.

Figure 6 defines the client-server of our running example using our PDDL domain. It extensively states the facts that are true in the architecture. The facts that are not listed are assumed false. This conjonction of facts conforms to the constraints that we have identified in the PDDL domain.

```
(:goal (and (exist_component Client)
            (exist_component BackupServer)
            (exist_connector Conn)
            (contains_component ClientServer Client)
            (contains_component ClientServer BackupServer)
            (contains_connector ClientServer Conn)
            (has_component_type Client Client_type)
            (has_component_type BackupServer BackupServer_type)
            (has_connector_type Conn Conn_type)
            (has_port Client request)
            (has_port BackupServer service)
            (has_role Conn requestor)
            (has_role Conn servicer)
            (type_has_port Client_type request)
            (type_has_port BackupServer_type service)
            (type_has_role Conn_type requestor)
            (type_has_role Conn_type servicer)
            (type_has_port ClientT request)
            (type_has_port BackupServer_type service)
            (bound Client request Conn requestor)
            (bound BackupServer service Conn servicer)))
```

Figure 7: The reconfigured client-server architecture in PDDL.

```
(add_port BackupServer BackupServer_type service)
(detach PrimServer service Conn servicer)
(create_component ClientServer BackupServer)
(attach BackupServer service Conn servicer)
```

Figure 8: The generated reconfiguration plan.

Following the same principle, Figure 7 defines the client-server of our running example after the reconfiguration. This is the goal clause of the PDDL problem file. With this example, the generated plan is the one given in Figure 8:

1. First, a `service` port is added to the `BackupServer` component. Indeed this port is absent in the initial architecture of Figure 6.
2. Second, the `service` port of the `PrimServer` component is detached from the `servicer` role of the `Conn` connector.
3. Third, the `BackupServer` component is instantiated within the `ClientServer` system.
4. Last, the `service` port of the `BackupServer` component is bound to the `servicer` role of the `Conn` connector.

As the goal does not state that the `PrimServer` component should be destroyed, the generated plan does not executes the `remove-component` action. Except few differences as noticed, the generated plan is almost the same as the handwritten reconfiguration of Figure 2. According to the specification of the actions, the `add-port` and `create-component` actions can be executed in any arbitrary order.

Architectural styles are translated as additional constraints in the problem file.

6. CHECKING INVARIANTS

During a reconfiguration, the software system passes by a succession of intermediate architectures, at the end of each

```
(:constraints (always (and
 (forall (?c1 ?c2             - component
          ?co                 - connector
          ?request ?service   - port
          ?requestor ?servicer - role
          ?c1T ?c2T           - component-type
   (implies
     (and (has-port ?c1 ?request)
          (has-port ?c2 ?service)
          (has-role ?co ?requestor)
          (has-role ?co ?servicer)
          (bound ?c1 ?request ?co ?requestor)
          (bound ?c2 ?service ?co ?servicer)
          (has-component-type ?c1 ?c1T)
          (has-component-type ?c2 ?c2T))
     (and (forall (?p - port) (implies (type-has-port ClientT ?p)
                                       (type-has-port ?c1T ?p)))
          (forall (?p - port) (implies (type-has-port ServerT ?p)
                                       (type-has-port ?c2T ?p)))))))))
```

Figure 9: The client-server invariant of Figure 2 as a PDDL trajectory constraint.

primitive reconfiguration operation. Several invariants must hold in any architecture: some of them are inherent of the ADL itself; some of them come from the architectural style. Even if we choose to relax these constraints temporarily during reconfiguration, we have to ensure that the still-required properties are always satisfied. For instance, Fractal [7] and André *et al.* [1] forbid a component to be active if any of its client ports is not bound. Depending on the implementation, some orderings of the ACME `Detach` / `Attachments` might be forbiden as well, e.g., to prevent a client from being temporarily bound to two servers at the same time.

We consider two strategies in order to enforce invariants.

On the one side, we can encode the invariants as trajectory constraints, which have been introduced in PDDL3 [15]. Constraints are used to prune the search space of the planner. The underlying logic is equivalent to a limited subset of LTL. Temporal operators can be used to constrain dynamic architectures. As ACME does not support dynamic architectures, temporal operators are not needed in order to encode invariants. For instance, the clause in Figure 9 gives the PDDL syntax in order to encode the invariant for the client-server style. It states that if any two components `?c1` and `?c2` are connected by a connector `?co`, then their respective types are included in `ClientT` and `ServerT`, respectively.

The Armani language for ACME invariants is based on the first-order predicate logic, restricted to quantification over finite sets only, like PDDL. The only restriction is therefore the ability to encode the Armani primitive functions using our PDDL predicates. For instance, in the list of primitive functions of [21], the `satisfiesType` function can be implemented as the verification that the element has the same subelements as declared in the type. However, in our system, we do not provide the necessary mechanism for the `declaresType` function. Indeed, we do not keep track of subtyping declarations in the PDDL encoding. We would need to define additional predicates in order to implement the `declaresType` function.

On the other side, some of the invariants can be checked

```
(:derived (unbound-port ?c - component ?p - port)
  (forall (?co - connector ?r - role)
    (not (bound ?c ?p ?co ?r))))
(:derived (unbound-role ?co - connector ?r - role)
  (forall (?c - component ?p - port)
    (not (bound ?c ?p ?co ?r))))
```

Figure 10: PDDL definition of derived predicates.

statically. Indeed, if the primitive reconfiguration operations are properly modelled, they should at least preserve the invariants coming from the ADL, whatever the context. Therefore, we aim at proving that primitive operations are consistent with the invariants. Let $\mathcal{I}$ be the invariant. Given an operation with parameters p, if the precondition $P(p)$ holds, then the invariant must still be satisfied after the effect in $[E(p)]\mathcal{I}$:

$$\mathcal{I} \Rightarrow \forall p, P(p) \Rightarrow [E(p)]\mathcal{I}$$

As a simple example, we may want to ensure that any port is bound to at most one role and that no action can infringe that rule. This property is formalized as:

$$\begin{array}{l}\forall c : \texttt{component}, \forall p : \texttt{port},\\ \forall r_1, r_2 : \texttt{role}, \forall co_1, co_2 : \texttt{connector},\\ \texttt{bound}(c, p, co_1, r_1) \wedge \texttt{bound}(c, p, co_2, r_2)\\ \Rightarrow co_1 = co_2 \wedge r_1 = r_2\end{array}$$

Any operation that does not have any `bound` positive effect obviously preserves this invariant. When we check the `attach` operation, the positive effect $\texttt{bound}(\texttt{?c}, \texttt{?p}, \texttt{?co}, \texttt{?r})$ is established only if the precondition $\texttt{unbound-port}(\texttt{?c}, \texttt{?p})$ is true. The invariant holds as our predicates are such that $\forall c, \forall p, (\texttt{unbound-port}(c, p) \Leftrightarrow \forall co, \forall r, \neg\texttt{bound}(c, p, co, r))$.

Of course, the system has to be checked against this property and all other constraints of section 5.1 as well.

Only invariants that are general, at the level of the architecture description language, can be verified statically. Indeed, we involve solely the PDDL domain, i.e., the specification of predicates and reconfiguration actions, which is common to all of the possible architectures.

7. CONCLUSION

In this paper, we propose a schema to encode an ACME architecture using the PDDL language. This work has several interests. First, as pointed out by related works, the approach allows using automatic planner from the AI community in order to generate automatically reconfiguration scripts. Second, it lets us study reconfiguration operations that do not exist currently in Plastik, e.g., operations that affect the type of architectural elements. Third, we give a sound semantics to our reconfiguration framework.

In comparison to related works, we improve the technique in that we propose how to manage the type of architecture elements, as well as architectural styles and constraints. We furthermore formally state consistency constraints for our PDDL domain. We explain how all of these constraints can either be checked statically or used to restrict the state space of the planner.

In this paper, we do several design choices:

- We decide not to use PDDL derived predicates, i.e., predicates that are given as a formula of the other predicates. We can use this feature for instance for the `unbound−port` and `unbound−role` predicates. Using the definitions of Figure 10, we do not have to explicitly manage these predicates in the effects of `attach` and `detach`. We can implement the `exist−component` and `exist−connector` predicates as derived predicates using `contains−component` and `contains−connector` as well. However, while derived predicates are appealing, only few planners support this fragment of the PDDL language.

- We translate straightforwardly the invariants of the software architecture into PDDL constraints. As a consequence, the invariants must hold during the whole reconfiguration plan; and no reconfiguration action can change the set of enforced invariants.

- We assign a unique type to each component or connector. That way, we can safely elude the question: what happens to its type and to the components (resp. connectors) that share the same type when a port (resp. role) is added or removed to a component (resp. connector).

- We ignore some fragments of the ACME architecture description language. We do not consider properties, representations, type satisfaction declarations, families. Representations and type satisfaction declarations are relations between architectural elements; families are types for systems. We can therefore use the same approach as for the other kinds of relations. The properties store values of primitive types (integer, floating-point number, string, boolean), enumerated types or constructed types (sequence, set, record). Storing values is supported by PDDL fluents, which are functions that map a value to PDDL objects. However, fluents are restricted to either objects (which may suit well, e.g., enumerated types) or numbers. Instead of fluents, we can use the schema of El Maghraoui *et al.* [13].

This work is still in early stage. As short-term future work, we plan to experiment our PDDL domain with real planners. Indeed, it is well-known that almost no existing planner implements the whole PDDL standard, as reported at the International Planning Competition'2011. In our preliminary results using 65 planners[1], only 17 planners pass the running example of this paper (without using constraints). Among them, 14 planners generate the 4 actions plan of Figure 8[2]; the 3 other planners generate additional useless actions. Only 1 planner succeeds when we use derived predicates; no planner supports constraints. Among the 48 failing planners, 6 planners report that they do not support negation in action preconditions. We still need to investigate the reason why the 42 remaining planners fail.

Even if more experiments confirm that no planner support all of the PDDL features that we use, we notice that: negative preconditions can be removed by the introduction of additional predicates and effects; derived predicates can be expanded like we do in this paper, e.g., for `unbound−port`; quantifiers can be expanded as quantification is over finite sets; constraints can be encoded into preconditions. Furthermore, these features are not implemented probably due to the fact that they are not used in the International Planning Competition. We can therefore expect that they will be supported as the competition evolves next years.

The performance of planners is also affected by some metrics in the domain definition such as maximum number of positive or negative effects and preconditions [8]. We therefore have to ensure that existing planners behave correctly with our improvements. During our first experiments, simple reconfigurations are solved in less than 300ms with an Intel E5400 / Linux x64 PC, except one planner that takes 2.5s. In our future work, we will evaluate the planners with more complex reconfigurations.

[1] 55 competing planners and 3 non competing planners come from the International Planning Competition'2011 public subversion repository; 7 planners are downloaded from public web sites. Some of these 65 planners differ only in the heuristics and parameters they use.

[2] Some planners generate a different sequence, but the actions are the same. Some planners propose to execute several of these actions in parallel. We have verified that all of the generated solutions are correct.

8. REFERENCES

[1] F. André, E. Daubert, G. Nain, B. Morin, and O. Barais. F4Plan: an approach to build efficient adaptation plans. In *7th International ICST Conference on Mobile and Ubiquitous Systems*, Sydney, Australia, Dec. 2010.

[2] J. Appavoo, K. Hui, C. Soules, R. Wisniewski, D. D. Silva, O. Krieger, D. Edelsohn, M. Auslander, B. Gamsa, G. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, and J. Xenidis. Enabling autonomic behavior in systems software with hot-swapping. *IBM Systems Journal*, 42(1), Jan. 2003.

[3] J. Armstrong. *Programming Erlang: software for a concurrent world*. The Pragmatic Bookshelf, 2007.

[4] N. Arshad and D. Heimbigner. A comparison of planning based models for component reconfiguration. Technical Report CU-CS-995-05, University of Colorado, Boulder, Colorado, USA, 2005.

[5] N. Arshad, D. Heimbigner, and A. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. *Software Quality Journal*, 15(3):265–281, Sept. 2007.

[6] T. Batista, A. Joolia, and G. Coulson. Managing dynamic reconfiguration in component-based systems. In *Software Architecture*, volume 3527 of *LNCS*, pages 439–480, Pisa, Italy, June 2005.

[7] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The FRACTAL component model and its support in Java. *Software: Practice and Experience*, 36(11-12):1257–1284, Sept. 2006.

[8] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1–2):165–204, Sept. 1994.

[9] A. Coles, A. Coles, M. Fox, and D. Long.

Forward-chaining partial-order planning. In *International Conference on Automated Planning and Scheduling*, pages 42–49, Toronto, Ontario, Canada, May 2010.

[10] P.-C. David and T. Ledoux. Safe dynamic reconfigurations of Fractal architectures with FScript. In *Proceedings of the 5th Fractal Workshop at ECOOP*, Nantes, France, July 2006.

[11] M. Dmitriev. *Safe class and data evolution in large and long-lived Java applications*. PhD thesis, University of Glasgow, Mar. 2001.

[12] P. Duquesne and C. Bryce. A language model for dynamic code updating. In *International Workshop on Hot Topics in Software Upgrades*, Nashville, Tennessee, USA, Oct. 2008.

[13] K. El Maghraoui, A. Medhranjani, T. Ailam, M. Kalantar, and A. Konstantinou. Model driven provisioning: bridging the gap between declarative object models and procedural provisioning tools. In *Middleware*, volume 4290 of *Lecture Notes in Computer Science*, pages 404–423, Melbourne, Australia, Nov. 2006.

[14] D. Garlan, R. Monroe, and D. Wile. Acme: an architecture description interchange language. In *CASCON First Decade High Impact Papers*, pages 159–173, 2010.

[15] A. Gerevini and D. Long. Plan constraints and preferences in PDDL3. Technical Report RT 2005-08-47, Università degli Studi di Brescia, 2005.

[16] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL – the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, Oct. 1998.

[17] K. M. Hansen and M. Ingstrup. Modeling and analyzing architectural change with Alloy. In *Symposium on Applied Computing*, pages 2257–2264, Sierre, Switzerland, Mar. 2010.

[18] M. Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[19] IBM. Tivoli provisioning manager. `http://www-01.ibm.com/software/tivoli/products/prov-mgr/`.

[20] M. Ingstrup and K. M. Hansen. Modeling architectural change: architectural scripting and its applications to reconfiguration. In *European Conference on Software Architecture*, pages 337–340, Cambridge, UK, Sept. 2009.

[21] R. Monroe. Capturing software architecture design expertise with Armani. Technical Report CMU-CS-98-163, Carnegie Mellon University School of Computer Science, Jan. 2001.

[22] B. Morin, O. Barais, J.-M. Jézéquel, B. Surajbali, G. Blair, A. Rashid, and N. Bencomo. Diva reference architecture. Technical Report D3.3, DiVA, Aug. 2010.

[23] B. Morin, F. Fleurey, N. Bencomo, J.-M. Jézéquel, A. Solberg, V. Dehlen, and G. Blair. An aspect-oriented and model-driven approach for managing dynamic variability. In *Model driven engineering languages and systems*, volume 5301 of *LNCS*, pages 782–796, Toulouse, France, Oct. 2008. Springer.

[24] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oril. Practical dynamic software updating for C. In *ACM SIGPLAN conference on Programming Language Design and Implementation*, Ottawa, Canada, June 2006.

[25] S. Richter and M. Westphal. The LAMA planner: guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39:127–177, 2010.

[26] J. Rintanen. Planning with specialized SAT solvers. In *AAAI Conference on Artificial Intelligence*, pages 1563–1566, San Francisco, California, USA, Aug. 2011.

[27] M. Wermelinger, A. Lopes, and J. L. Fiadeiro. A graph based architectural (re)configuration language. In *European Software Engineering Conference*, pages 21–32, Vienna, Austria, Sept. 2001.