PLASMA: A Plan-based Layered Architecture for Software Model-driven Adaptation

Hossein Tajalli, Joshua Garcia, George Edwards, and Nenad Medvidovic
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781, USA
{tajalli,joshuaga,gedwards,neno}@usc.edu

ABSTRACT

Modern software-intensive systems are expected to adapt, often while the system is executing, to changing requirements, failures, and new operational contexts. This paper describes an approach to dynamic system adaptation that utilizes plan-based and architecture-based mechanisms. Our approach utilizes an architecture description language (ADL) and a planning-as-model-checking technology to enable dynamic replanning. The ability to automatically generate adaptation plans based solely on ADL models and an application problem description simplifies the specification and use of adaptation mechanisms for system architects. The approach uses a three-layer architecture that, while similar to previous work, provides several significant improvements. We apply our approach within the context of a mobile robotics case study.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.11 [**Software Engineering**]: Software Architecture

General Terms

Design, Management, Reliability

Keywords

Self-*, software adaptation, software architecture, software evolution, planning, model-driven software, component-based software

1. INTRODUCTION

Modern software systems are increasingly expected to satisfy high reliability and availability requirements. In particular, these systems are expected to alter and extend their functionality, handle failures of software and hardware components, and apply updates and bug fixes while the system is executing.

To provide these capabilities, several software architecture-based self-adaptive and self-aware systems [6, 12, 21, 9] have been proposed in the literature. A software architecture-based approach to dynamic system adaptation has several advantages. Architecture-based approaches utilize composition, hierarchy, and abstraction to reduce complexity and increase scalability. Furthermore, a rich area of existing work on software architecture modeling and analysis can be exploited. For example, architecture description languages (ADLs) [23] can be used to specify a system that must undergo runtime adaptation, and architectural analysis techniques can be used to examine the implications of system adaptations.

Previous work in self-adaptive and self-managing systems can be categorized across two prevalent adaptation mechanisms: policies and plans. Policies are sets of condition-action rules that specify how the system should be modified when a specific condition is met. An example is the policy-based approach to architectural adaptation management (PBAAM) [9]. On the other hand, plans are sequences of actions that achieve a high-level goal. Plans are automatically generated by finding a path from the current system state to a goal state in a model of the system domain. Sykes et al.'s three-layer model [21] is an example of a plan-based approach.

Thus far, plan-based approaches have been limited to pre-computed plans for application adaptation and have lacked the ability to dynamically generate new plans when system requirements change. Furthermore, planning mechanisms have only been applied to core application functionality, and have not been applied to application architecture adaptation. These approaches are, thus, unable to dynamically compute new adaptation plans that are used to modify application *architectures* in response to unforeseeable events, such as new system requirements, a change in the execution environment, or an unexpected type of failure.

In this paper, we introduce PLASMA (a Plan-based Layered Architecture for Software Model-driven Adaptation). PLASMA utilizes an ADL and a planning-as-model-checking technology [10] to enable dynamic replanning in the architectural domain. We utilize ADL models and system goals as inputs for generating plans. In particular, our solution has the ability to generate new plans in response to changing system requirements and goals and in the case of unforeseeable component failures.

The contributions of our paper are twofold. First, we show how planning techniques can be applied not only to application functionality, but to application architecture adaptation, which results in the ability to automatically modify application architectures in order to achieve a system goal when the current architecture is inadequate. Second, we show how dynamic replanning can be used when the application domain or system goal changes, which results in the ability to automatically handle unexpected circumstances.

The rest of the paper is organized as follows. Section 2 describes our framework's architecture and PLASMA's use of domain models, component models, and planning. Section 3 details PLASMA's design and implementation. Section 4 demonstrates how we ap-

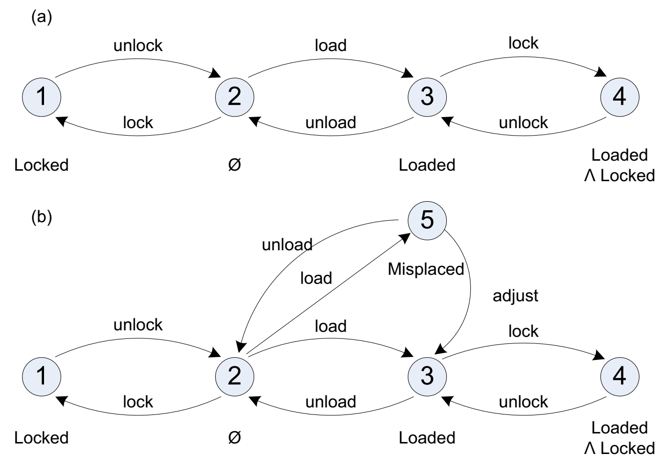


Figure 1: (a) An example application domain model. (b) The example extended with an additional capability.

plied PLASMA in a mobile robotics case study and discusses the benefits observed. Section 5 covers the related work , and Section 6 concludes the paper.

2. APPROACH

In this section, we provide a high-level summary of the PLASMA approach to automated self-adaptation and self-management for software systems; the details of PLASMA's design and implementation are then provided in Section 3. First, we explain the two types of domain models that are utilized in PLASMA: application domain models and adaptation domain models. Second, we describe how PLASMA leverages an adaptive layered architecture to achieve a high-degree of autonomy and a clear separations of concerns. Third, we discuss the role of architectural modeling in PLASMA. Lastly, we describe how PLASMA applies planning techniques to realize dynamic replanning in a self-adapative system.

2.1 Domain Models

PLASMA is based on the novel application of planning mechanisms to two different types of models: *application domain models* and *adaptation domain models*. Application domain models capture the possible states of application components and actions that those components may perform. Each action modifies the application state in a defined way. Analogously, adaptation domain models capture architectural states and actions: each state in the adaptation domain model corresponds to a particular architectural configuration, and actions in the adaptation domain model correspond to architectural modifications, such as the addition, removal, and replacement of components. While previous work has applied planning to application domains, PLASMA is unique in its usage of adaptation domains.

Both application domain models and adaptation domain models consist of the *states* of the domain, available *actions* in the domain, and the *state transitions* caused by those actions. Therefore, a domain model is formally defined as a 4-tuple <**F**,**S**,**A**,**R**> in which, **F** is a finite set of fluents which represent the state variables of the system, $\mathbf{S} \subseteq 2^{\mathbf{F}}$ is a finite set of states, **A** is a finite set of actions and $\mathbf{R} : \mathbf{S} \times \mathbf{A} \longmapsto \mathbf{S}$ is a transition function.

Figure 1 depicts two example application domain models. The first application domain model represents a system where an item can be loaded/unloaded to/from a container which, in turn, can be locked/unlocked. Figure 1(a) is a graphical presentation of the domain model for this example in which *Loaded* and *Locked* are the fluents of the domain, circles represent the states, and arrows represent the actions as well as system transitions in the model. The label assigned to each state in this figure is the conjunction of the fluents which are true in that state. For the case where a system requirements change occurs, consider the domain model depicted in Figure 1(b). In this figure, a new fluent called *Misplaced* and a new state are added to handle the case where an item may be misplaced.

Figure 2 depicts an example adaptation domain model. In this example domain, software components can be instantiated/killed, added/removed to/from an architecture, and connected/disconnected to/from another component in that architecture. The states of this adaptation domain model are the various architectural configurations involving C1 and C2. For example, state 7 corresponds to an architecture in which C1 and C2 exist but are not connected.

In the remainder of this section, we show how these two types of domain models are utilized for software adaptations in PLASMA.

2.2 Adaptive Layered Architecture

In our approach, we leverage the *adaptive layered architectural style* which was introduced in our previous work [5]. Traditionally, layering implies that components at a given layer *invoke the services of* components at the layer below. In contrast, components at a given layer in the adaptive layered style *monitor, manage, and adapt* components at the layer below. Adaptive layered systems consist of application-level components and meta-level components. Application-level components implement functionality that achieves the application goals. Meta-level components are designed to handle operations that deal with monitoring, analysis, and adaptation. Meta-level components can be one of three types: *Collectors*, *Analyzers*, or *Admins*. *Collectors* monitor components, *Analyzers* evaluate adaptation policies or plans based on monitored data, and *Admins* actually modify components.

Although the adaptive layered style allows for layering of arbitrary depth, PLASMA employs three adaptive layers, as shown in Figure 3. Application-level components reside in the bottom *application layer*. The middle layer, called the *adaptation layer* monitors, manages, and adapts components in the application layer. The topmost *planning layer* manages the adaptation layer and the generation of plans based on user-supplied goals and component specifications. The plans generated by the planning layer define both the target architecture for the application layer (the *adaptation plan*) and the actions to be carried out by the application layer (the *application plan*). Therefore, the planning layer is capable of responding to changing system requirements or operational environments by regenerating plans. This three-layered architecture enforces a clear separation of concerns, whereby each layer in the system provides a different form of adaptation capability, and enables a high-degree of autonomy.

In PLASMA, the only inputs provided by the system architect are (1) the application problem description, (2) component specifications written in an ADL, and (3) executable implementations of the components. The problem description consists of an initial state and a goal. The component ADL models are used by the *ADL Model Parser* components in the planning layer to generate application and adaptation *domain model descriptions*, as depicted in Figure 3. A domain model description presents a domain model in a standard language which is required for planning. Domain model descriptions, along with the problem description, are provided to the *Application Planner* and *Adaptation Planner* components depicted in the planning layer in Figure 3. Each of these two components generates a plan for one of the two bottommost lay-

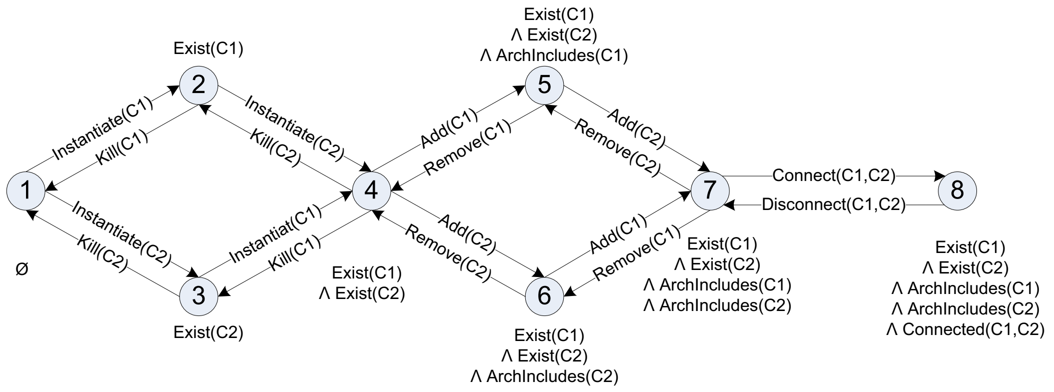


Figure 2: An example adaptation domain model.

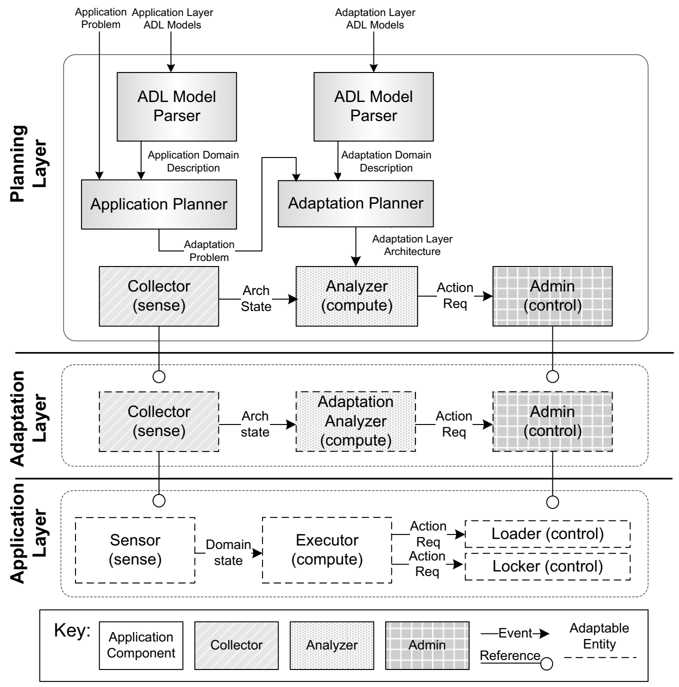


Figure 3: The PLASMA adaptive layered architecture.

ers: the *Application Planner* generates a plan for the application layer which specifies how to achieve the application goal, while the *Adaptation Planner* generates a plan for the adaptation layer which specifies how to arrive at the target application architecture.

Each of the two bottommost layers contain a special component that executes the plans generated by the top layer's planner components. The adaptation layer contains an *Adaptation Analyzer* that executes plans generated by the *Adaptation Planner*. *The Adaptation Analyzer* uses the plan to determine how and under what conditions components in the application layer should be added, removed, replaced, or otherwise altered. The application layer contains a special component called the *Executor*, which executes plans generated by the *Application Planner*. The *Executor* examines the state of application components and, based on their state and the plan received from the *Application Planner*, invokes the functionality of application components to perform tasks.

2.3 Architectural Modeling

As previously noted, components in PLASMA must be specified in an ADL and passed to the *ADL Model Parsers* in the topmost layer in Figure 3. ADL models are used to generate domain model descriptions which are required for planning. To be able to generate domain models from ADL models, the ADL used must carry enough information from which a 4-tuple domain model, <**F**,**S**,**A**,**R**>, can be extracted. Consequently, component descriptions in the selected ADL should include the ability to specify (1) attributes, (2) required and provided interfaces, and (3) pre- and post-conditions of each interface. This information is used by the *ADL Model Parsers* to generate domain model descriptions which are passed to the appropriate *Planner* component. In particular, the *ADL Model Parser* will generate a domain model by mapping attributes to fluents **F**, interfaces to actions **A** and by utilizing pre- and post-conditions of interfaces to determine states **S** and the transition function **R**. Instead of explicitly (and possibly rigidly) specifying the structure of the application's architecture, PLASMA infers the topology of the application layer from the plan and the component ADL models. To this end, required and provided interfaces are used to discover component dependencies and, thus, determine required components in the architecture. We should point out that any ADL that includes this information can be used with PLASMA, given the appropriate *ADL Model Parser*.

The use of ADL models in PLASMA results in several benefits. First, whenever components are added, removed, or replaced, new plans can be automatically generated to achieve the goals of the system. In particular, ADL model changes result in domain model changes, which in turn, initiate replanning. Second, changes to requirements represented in changes to component model descriptions can be handled during runtime. This particular benefit is obtained because the domain models that are generated from ADL models serve as a requirements specification of the system.

In the next section, we describe how the planners in Figure 3 use domain model descriptions to create plans for both the application and adaptation layers.

2.4 Planning

The behavior of the application and adaptation layers in PLASMA are planned in order to achieve the high level goal of the application layer, specified by the system architect (i.e., the PLASMA user). The two bottommost layers receive the plans from the planning layer. The planning layer generates different kinds of plans for the application and adaptation layers. Each of these plans is generated from its own domain model description. The planner components in the top layer accept the domain model description and a planning problem as inputs to generate a plan. We maintain a clear separation of application and adaptation concerns by separating planning among the two different kinds of planners in the planning layer.

A planning problem is the problem of arriving at a goal from an initial state. The goal is represented by a set of conditions on the domain model fluents, and corresponds to a set of goal states where those conditions are true. The generated plan is a set of state-action rules which specify the actions to be taken from each state to reach a goal state. The plan is sufficient to determine a path from any possible initial state to a goal state.

The application goal is a set of conditions provided by the system architect. The *Application Planner* first finds an application plan that specifies how to achieve the application goal from a given current application state. The *Application Planner* then derives the target architecture topology of the application layer required to run that plan, using an algorithm specified in Section 3. The current architectural topology and the target architecture topology of the application layer form the inital and goal states of the *Adaptation Problem*, respectively. Consequentely, the *Adaptation Planner* finds an adaptation plan that transforms the current architecture of the application layer to the desired architecture.

The application plan and adaptation plan are executed in the following way. Recall that the adaptive layered style includes meta-level components—*Collectors*, *Analyzers*, and *Admins*—that respectively monitor, analyze, and modify components in the layer below (see Figure 3).

1. The *Collector* at the planning layer determines the current architecture of the adaptation layer (which is initially empty). The planning layer *Analyzer* computes an architecture for the adaptation layer (i.e., what *Collectors*, *Analyzers*, and *Admins* are needed). The planning layer *Admin* instantiates and deploys the adaptation layer components. Unlike the application layer, the topology of the adaptation layer is provided to the *ADL Model Parser* component in the planning layer (i.e., it is not planned, but is rather pre-defined as shown in Figure 3).

2. The instantiated adaptation layer architecture will include an *Adaptation Analyzer* that contains the adaptation plan and logic for executing it. The *Adaptation Analyzer* executes the plan by instructing the adaptation layer *Admin* to make modifications to the application layer architecture by instantiating components, establishing connections, etc. Adaptation layer *Collectors* monitor the application architecture; if it changes unexpectedly, the *Adaptation Analyzer* can use the adaptation plan to return to the target application architecture.

3. The instantiated application layer architecture will include an *Executor* that contains the application plan and logic for executing it and one or more *Sensors*. *Sensors* monitor the current state of the system (i.e., the domain fluents) and pass this information to the *Executor*. The *Executor* uses the values of the fluents to determine the appropriate action to perform, according to the application plan.

Adaptation in our approach is achieved automatically through dynamic replanning and is not pre-programmed by the system architect. Dynamic replanning can be initiated either by the system architect or by unexpected changes to the domain models (e.g., due to component failures). Adaptation of the application layer occurs whenever the adaptation goal and/or adaptation domain model

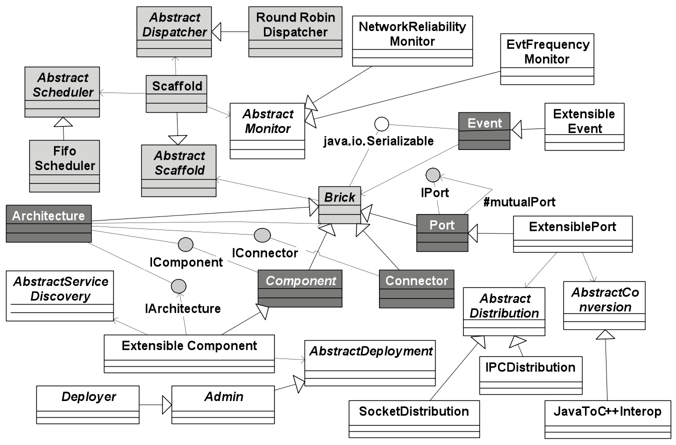


Figure 4: The high-level design of Prism-MW.

change. These changes could, in turn, be caused by a change to the application goal and/or the application domain model. The first case (application goal change) is initiated by the system architect, who may provide a new application goal to the system at any time. In the second case (application domain model change), a new component becomes available (e.g., a new version is introduced) or a component becomes unavailable (e.g., due to a failure).

The planning and adaptation layers achieve a high degree of autonomy and flexibility, and also reduce the burden on the architect. The adaptation layer maintains a high degree of autonomy by utilizing automatically generated plans to adapt the application layer. Placing the onus of automatic adaptation onto the adaptation layer reduces the burden of design on the architect because she need not manually specify plans for each adaptation scenario. Moreover, our approach offers significant flexibility by giving the architect the ability to specify the adaptation requirements via ADL models of specific components in the adaptation layer.

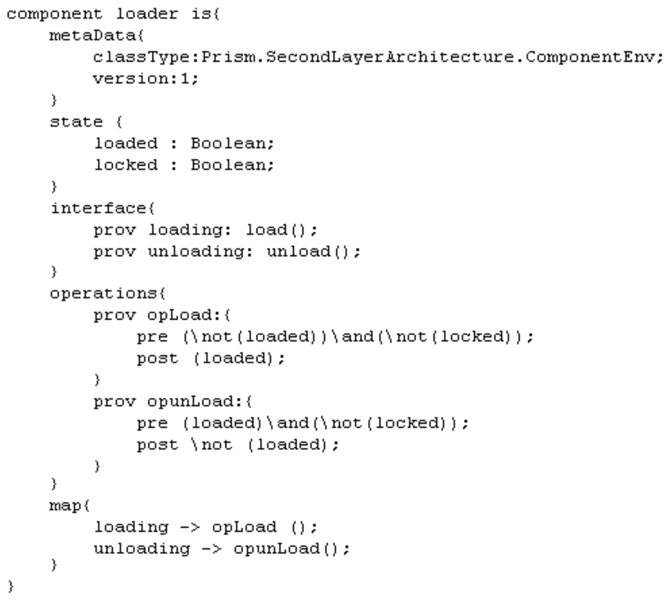
3. DESIGN AND IMPLEMENTATION

In this section we describe the detailed design and implementation of PLASMA. In particular, we discuss the middleware, ADL, and ADL tool support we utilized, as well as the tools we leveraged from the planning domain.

3.1 Adaptive Layered Architectures in PLASMA

As mentioned in 2.2, we leveraged the adaptive layered style introduced in our previous work [5]. As described in [5], we extended an *architectural middleware* platform called Prism-MW [13] to support adaptive layered architectures. Prism-MW is a lightweight middleware implemented in Java and C++. The high-level design of Prism-MW is given in Figure 4. The key building blocks of Prism-MW are the *Architecture*, *Component* and *Connector* classes. An *Architecture* object serves as a run-time container for a topology of *Components* and *Connectors*. *Components* implement application services, while *Connectors* implement interaction-oriented services. *Components* and *Connectors* communicate by exchanging *Events* via *Ports*. Prism-MW is particularly well-suited to PLASMA because it natively supports dynamic system adaptation by decoupling components via (dynamically instantiable) ports and event-based interaction. The full description of this platform is beyond the scope of the current paper and may be found in [13].

The extended version of Prism-MW which is introduced in [5]



```
component loader is{
    metaData{
        classType:Prism.SecondLayerArchitecture.ComponentEnv;
        version:1;
    }
    state {
        loaded : Boolean;
        locked : Boolean;
    }
    interface{
        prov loading: load();
        prov unloading: unload();
    }
    operations{
        prov opLoad:{
            pre (\not(loaded))\and(\not(locked));
            post (loaded);
        }
        prov opunLoad:{
            pre (loaded)\and(\not(locked));
            post \not (loaded);
        }
    }
    map{
        loading -> opLoad ();
        unloading -> opunLoad();
    }
}
```

Figure 5: The SADEL specification for the *Loader* component.

includes three specialized meta-level component types: *Collectors*, *Analyzers*, and *Admins*. Meta-level components have access to the internal representation of a currently running architecture. We call components with this kind of access *architecturally aware. Collectors* and *Analyzers* have read-only access to the architecture, while *Admins* have read/write access and are allowed to adapt the architecture. Moreover, the extended version of Prism-MW defines a *meta-level architecture* type which is used as a container for the meta-level components. Each meta-level component in a meta-level architecture possesses a reference to the architecture that implements the layer immediately below it.

We evolved and extended the meta-components of the adaptive layered style to run application and adaptation plans in PLASMA. As envisioned in [5], *Analyzers* evaluate the adaptation policies based on monitoring data and triggered adaptation operations when required. These policies and plans were designed and implemented by application developers and were hard-coded in the *Analyzer*. Therefore, *Analyzers* were not very flexible and did not provide a high degree of automation. Furthermore, they tended to be complex, which made them time consuming and error-prone to generate. In contrast, the *Adaptation Analyzer* in PLASMA is an extended version of the one in [5]; it is capable of running automatically generated plans. We also modified the *Collector* and *Admin* components to match the interfaces of our *Adaptation Analyzer*.

3.2 Architectural Modeling in PLASMA

To support architectural modeling in PLASMA, we adapted and extended C2SADEL, an ADL introduced in [15]. We chose this language because it meets the requirements discussed in 2.3. Moreover, it supports system analysis and evolution. This language was originally designed to support the C2 architectural style, a layered and event-based style. Furthermore, we modified the component-based DRADEL environment [15] used to support modeling, analysis, evolution, and implementation of architectures described in C2SADEL. DRADEL supports component evolution via subtyping, architectural consistency- and type-checking, and code generation.

We extended C2SADEL and created SADEL as the ADL for PLASMA. As discussed in 3.1, Prism-MW supports the implementation of the layered component-based architecture of PLASMA. Therefore, we designed SADEL to support the event-based style which is the basic style supported in Prism-MW. Compared to C2, the event-based style covers a wider range of applications and provides a higher degree of flexibility in architectural design (though at the cost of some analyzability, as discussed in [23]). Just like in C2SADEL, an architectural model in SADEL has three major parts: *component* types, *connector* types and a *topology*. The topology defines the component and connector instances and their interconnections. Components define state variables, interfaces, and provided and required operations. Each interface is mapped to an operation. In turn, each operation specifies the pre- and post-conditions of its invocation in terms of first-order logic expressions involving state variables. Operation definitions in SADEL specify conditional and non-deterministic post-conditions, which are required for the generation of domain models. Component models in SADEL also include version information, which allows the planning layer to distinguish between evolved/updated components and adapt the architecture accordingly.

An example SADEL specification of the *Loader* component is given in Figure 5. *Loader* has two interfaces: `load()` and `unload()`. These interfaces are mapped to actions **A** in the application domain model. *Loader* has two Boolean state variables called `loaded` and `locked`. These variables are mapped to fluents **F** in the domain model. Finally, pre- and post-conditions are specified for the `load()` and `unload()` interfaces. For example, `load()` has the pre-condition `(\not(loaded))\and(\not (locked))` and post-condition `loaded`.

PLASMA provides support for modeling, analysis, evolution and implementation of components and architectures described in SADEL by enhancing DRADEL [15] and integrating it with other PLASMA components. We ported DRADEL's components to PRISM-MW and adapted their interfaces to match PLASMA components in the planning layer. The enhanced DRADEL components integrated into PLASMA support modeling and analysis in the event-based (as opposed to C2) style and code generation for PRISM-MW (as opposed to C2's custom-built architecture implementation framework). Incorporation of the extended DRADEL components enables PLASMA to perform consistency- and type-analysis on application layer architectures; further explanation of DRADEL's analysis capabilities can be found in [15]. These components also provide additional features not supported by DRADEL originally. For example, PLASMA supports component evolution via subtyping and version control.

3.3 Planning in PLASMA

We used the *Model Based Planner* (MBP), a planning tool for non-deterministic domains [3], to support planning in PLASMA. MBP implements the planning-as-model-checking technique introduced in [10]. The fundamental idea behind planning-as-model-checking is to generate plans by checking the correctness of formulas in a model. To find a plan to reach a goal state in a domain, MBP uses a problem description and a domain model description specified in NPDDL, an extended planning domain description language. The problem description determines what kind of plan (Weak, Strong, Strong Cyclic, etc. [10]) is required for the applicaion.

In the rest of this section, we describe how planning occurs at the application and adaptation layers. More specifically, we describe how PLASMA deals with initialization of architectures, topology determination, and goal or ADL model changes.

```
(define (domain test)
    (:predicates
        (loaded) (locked)
    )
    (:action load
        :precondition   (and(not(loaded))(not(locked)))
        :effect         (loaded)
    )
    (:action unload
        :precondition   (and(loaded)(not(locked)))
        :effect         (not(loaded))
    )
    (:action unlock
        :precondition   (locked)
        :effect         (not(locked))
    )
    (:action lock
        :precondition   (not(locked))
        :effect         (locked)
    )
)
```

Figure 6: An NPDDL domain model description for the example domain.

3.3.1 Application Planning

In PLASMA, the NPDDL application problem description (initial state and goal) is provided by the user, while the NPDDL application domain model description is created automatically from the SADEL models of application components by the *ADL Model Parser* (recall Figure 3). For illustration, the NPDDL specification of the application domain model depicted in Figure 1(a) is shown in Figure 6. This NPDDL spec is derived from the SADEL models of the *Loader* and *Locker*. For example, the `loaded` and `locked` state variables of the *Loader* in Figure 5 are mapped to the `loaded` and `locked` predicates in Figure 6, while the `load()` and `unload()` interfaces are mapped to the `load` and `unload` actions. Similarly, the NPDDL pre-conditions and effects of these actions are mapped from the pre- and post-conditions of the respective SADEL interfaces.

Once the NPDDL application domain model has been generated, it is passed, along with the NPDDL problem description, to the *Application Planner*, which generates the application plan if there is a path from the initial state to the goal state in the application domain model. If there is no such path, PLASMA notifies the architect so that she can modify the problem or the set of avaliable components. In our example domain, assume an initial state in which no item is loaded in the container and the container is not locked ($\neg Loaded \wedge \neg Locked$). This state is shown as state 2 in Figure 1(a). Also assume that the application goal is to load an object into the container and lock it. This goal is represented as state 4 ($Loaded \wedge Locked$) in Figure 1(a). The *Application Planner* generates a plan in the form of a set of state-action rules which should be followed to arrive at the goal. In this example, the plan contains the following pairs: {{1,unlock} {2,load} {3,lock}}. This plan is interpreted as follows: to get to state 4 the following actions must occur—unlock action on state 1, load action in state 2, and lock action in state 3. Although this is a simple example, richer applications have much longer plans and may have multiple plans to reach a goal. PLASMA reduces the burden on the architect by determining these plans automatically.

Next, to be able to run the application plan in the application layer, the required components for the plan and their topology must be determined. The responsibility of determining the topology of the application layer is placed upon the *Application Planner* component. The *Application Planner* determines the components which are required to complete the plan by examining the actions used in the plan. The *Application Planner* computes all the components' possible dependencies by matching their required and provided interfaces and creates a topology according to those dependencies. As a result, the architect no longer bears the burden of specifying the topology of the system.

In the example domain from Figure 1(a), the *Application Planner* determines the target application layer architecture in the following way. As previously mentioned, to get to state 4 from state 2 the following plan $P1$ should be run: {{1,unlock}{2,load}{3,lock}}. The *Loader* and *Locker* have the `load()`, `lock()`, and `unlock()` interfaces that match the actions of $P1$, so these components are included in the architecture. Recall that the application layer always includes an *Executor* that runs the application plan by invoking the interfaces of application components, such as *Loader* and *Locker*. Consequently, the *Loader* and *Locker* are connected to the *Executor*. Finally, the *Executor* must know the values of domain fluents in order to determine the appropriate actions. Consequently a *Sensor* is added to the architecture to monitor each fluent, and each *Sensor* is connected to the *Executor*. This target architecture, produced by the *Application Planner*, is passed to the *Adaptation Planner* in the form of an adaptation problem (initial architecture and goal architecture).

An application plan runs in the following manner. The *Sensor* component tracks the domain fluents, i.e., `locked` and `loaded` in our example. Changes to these values are pushed to the *Executor*. The *Executor* uses the values of the fluents to determine the appropriate action to run based on the plan. The actions are performed by invoking the interfaces of either the *Loader* or *Locker* components. This process is repeated until a goal state is reached. A high degree of autonomy is maintained since the plan is run by the *Executor* without any involvement from the architect.

While the previous discussion describes plan execution in the case where the goal of the system remains the same, a change to the goal requires replanning. Application replanning followed by software adaptation occurs automatically in PLASMA. In Figure 1(a), the application layer goal can be changed from $(Loaded \wedge Locked)$ to $(\neg Loaded \wedge Locked)$. This change initiates replanning which results in the following application plan: {{4,unlock} {3,unload} {2,lock}}. The adaptation layer updates the *Executor* component in the application layer to run the new plan.

To reiterate, the application layer and its key components are designed to ease the burden of design on the architect, maintain a high degree of autonomy for the software system, and handle non-deterministic domains.

3.3.2 Adaptation Planning

Adaptation plans are generated by the *Adaptation Planner* in the planning layer. Adaptation plans describe how to initialize the architecture, which includes the creation and connection of components. The *Adaptation Planner* requires an NPDDL adaptation problem, generated automatically by the *Application Planner*, and an NPDDL adaptation domain description, specified by the system architect, to create an adaptation plan. Adaptation plans are also computed by MBP using the same algorithm used to compute application plans.

Figure 2 shows an example adaptation domain model with two components. In our implementation, we utilized meta-level components, *Admin* and *Collector*, to perform the actions and monitor the fluents in the adaptation domain model. More specifically in the specific example depicted in Figure 2, the *Admin* component performs the *Instantiate*, *Kill*, *Add*, *Remove*, *Connect*, and *Discon-*

```
(define (domain AdaptationModel)
    (:types Component )
    (:predicates
        (Connected ?comp1 - Component ?comp2 - Component)
        (ArchIncludes ?comp - Component)
        (Exist ?comp - Component)
    )
    (:action Instantiate
        :parameters      (?comp - Component)
        :precondition    (not (Exist ?comp))
        :effect          (Exist ?comp)
    )
    (:action Add
        :parameters      (?comp - Component )
        :precondition    (not (ArchIncludes ?comp))
        :effect          (ArchIncludes ?comp )
    )
    (:action Connect
        :parameters      (?comp1 - Component ?comp2 - Component)
        :precondition    (and (Exist ?comp1) (ArchIncludes ?comp1)
                         (Exist ?comp2)(ArchIncludes ?comp2 ) (not
                         (Connected ?comp1 ?comp2))
        :effect          (Connected ?comp1 ?comp2)
    )
    .
    .
    .
)
```

Figure 7: NPDDL domain model description for the example adaptation domain.

nect actions in the adaptation domain and the *Collector* detects the *Exist*, *ArchInclude*, and *Connected* fluents.

The adaptation plan is executed and managed by the *Adaptation Analyzer* from Figure 3. The *Adaptation Analyzer* executes an adaptation plan autonomously by reading fluents from the *Collector* in the layer it resides in and by sending appropriate adaptation actions to be performed by the *Admin*. Therefore, adaptation planning and adaptation take place automatically and the application architects only deal with the adaptation layer requirements and design. In situations where a new *Adaptation Analyzer* is needed, e.g., in the case of an adaptation plan change, the planning layer instantiates a new *Adaptation Analyzer* for the adaptation layer.

An adaptation plan in our running example executes in the following manner. The application layer is initially empty and the goal state of the application plan has a *Loader*, a *Locker*, and a *Sensor* whose interfaces are connected to the *Executor* component. Thus, the adaptation plan contains actions, such as *Instantiate(Locker)*, *Instantiate(Executor)*, *Instantiate(Sensor)*, *connect(Locker,Executor)* to connect component interfaces, and so on. Figure 3 gives the instance of our layered architecture after the adaptation plan was run and the application layer was set up.

Once an initial application layer architecture has been instantiated, two situations may result in adaptations to the application layer: modifications to the component SADEL models and component failures during runtime (which may result in changes to the NPDDL domain models). PLASMA supports these dynamic adaptations through replanning.

The first cause of replanning is a SADEL model change, which occurs when system requirements evolve. For example, to handle a new requirement to perform adjustments to the container, a new component called *Adjuster* is added to the system with an interface to perform the adjust action. The SADEL model is changed to reflect the addition of the *Adjuster* component. This SADEL model change generates a new NPDDL application domain model description, which initializes replanning. A new application plan is found automatically and the *Application Planner* from Figure 3 uses the new plan to derive a new application layer architecture that contains the *Adjuster* component. At this point, a new adaptation problem is generated and passed to the *Adaptation Planner*. In this adaptation problem, the initial state is the current application layer architecture and the goal is the new target architecture. Consequently, the *Adaptation Planner* creates a new adaptation plan which changes the architecture from the current topology to the new topology. This plan contains an action to add the *Adjuster*.

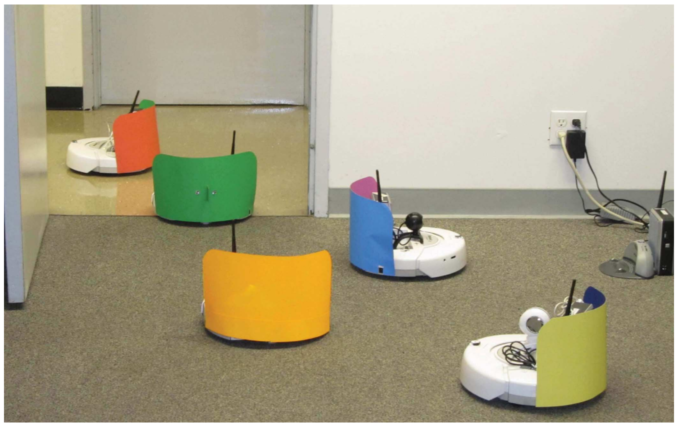


Figure 8: Robots convoying in the case study.

The second cause of replanning is a failure of an application component. As an example, the *Adjuster* component may fail at run-time. In this case, the *Executor* fails to successfully run the plan and the adaptation layer detects the component failure through its *Collector*. The *Adjuster* is removed from the ADL model, resulting in a change to the NPDDL application domain model description which, in turn, initiates replanning and the deployment of a new architecture that does not rely on the failed component. However, if a component failure makes it impossible to achieve the goal, PLASMA reports the failure to the architect.

PLASMA aims to reduce the burden of design placed upon the architect by preventing the architect from having to directly deal with architectural topology and adaptation plan specification. The architect does not need to anticipate all possible adaptation scenarios or pre-program adaptation plans. PLASMA is independent of the adaptation requirements and allows the user to provide any set of meta-level components for the adaptation layer. The meta-level components in the adaptation layer can be different for different adaptation domains, which provides further flexibility for the system's design. For example, a domain requirement can enforce disconnection of components in a particular order to ensure minimum disturbance to the system during runtime adaptation. PLASMA is able to natively include support for such a requirement.

4. CASE STUDY

In order to demonstrate the contributions and effectiveness of PLASMA, we have leveraged a family of robot-based systems developed in collaboration with a third-party organization [14]. In this section we first describe how component model changes, goal changes, and failures in a selected application scenario result in system adaptations.

4.1 Application Scenario

Our robotic application consists of three or more robots that form a convoy and follow a leader robot. The leader robot is provided with a path to follow in the form of a series of spatial coordinates called *waypoints*. Each follower robot uses on-board sensors to track the robot immediately ahead of it in the convoy and follow

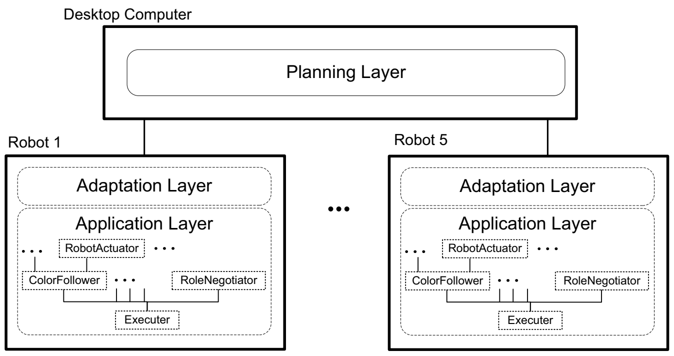


Figure 9: The deployment view of the case study architecture.

```
 ...
(case (and (= (BehindObstacle) 0) (= (CameraInUse) 0) (= (CanSeeIR) 1)
(= (CameraAvailable) 1) (= (EnoughLight) 1) (= (IsFollower) 1) (=
(IsLeader) 0) (= (IRInUse) 1) (= (IsFollowing) 0) (= (LeaderIsKnown)
0))
           (action (AssignALeader)))
(case (and (= (BehindObstacle) 1) (= (CameraInUse) 0) (= (CanSeeIR) 0)
(= (CameraAvailable) 1) (= (EnoughLight) 1) (= (IsFollower) 0) (=
(IsLeader) 1) (= (IRInUse) 0) (= (IsFollowing) 1))
           (action (StopLandmarkFollowing)))
(case (and (= (BehindObstacle) 1) (= (CameraInUse) 0) (= (CanSeeIR) 0)
(= (CameraAvailable) 1) (= (EnoughLight) 1) (= (IsFollower) 0) (=
(IsLeader) 1) (= (IRInUse) 0) (= (IsFollowing) 0))
           (action (AvoidObstacle)))
(case (and (= (BehindObstacle) 0) (= (CameraInUse) 0) (= (CanSeeIR) 1)
(= (CameraAvailable) 0) (= (IsFollower) 0) (= (IsLeader) 1) (=
(IsFollowing) 0))
           (action (StartLandmarkFollowing)))
(case (and (= (BehindObstacle) 0) (= (CameraInUse) 0) (= (CanSeeIR) 1)
(= (CameraAvailable) 1) (= (EnoughLight) 0) (= (IsFollower) 1) (=
(IsLeader) 0) (= (IRInUse) 0) (= (IsFollowing) 0) (=
(IRReceiverAvailable) 1) (= (LeaderIsKnown) 1))
           (action (StartIRFollowing)))
(case (and (= (BehindObstacle) 0) (= (CameraInUse) 0) (= (CanSeeIR) 0)
(= (CameraAvailable) 1) (= (EnoughLight) 1) (= (IsFollower) 1) (=
(IsLeader) 0) (= (IsFollowing) 0) (= (LeaderIsKnown) 1))
           (action (StartCameraFollowing)))
...
```

Figure 10: A snippet of the generated application plan for the robot following case study.

it. This type of autonomous convoy is a common use-case for mobile robotics, having applications in transportation [11], inventory management [24], automated farming [4], and other areas.

Each robot in our case study consists of an iRobot Create mobile programmable robot, an attached eBox 3854 embedded computer running Fedora Linux, a camera, a GPS receiver, an infrared (IR) receiver, and an IR transmitter. Also, the iRobot platform includes a front bumper sensor that detects when the robot has run into an obstacle, while the eBox includes a 802.11 wireless LAN adapter. A picture of the robots is shown in Figure 8.

Initially, we provided PLASMA with the SADEL models of the following components: *RoleNegotiator*, *WaypointFollower*, *CameraFollower*, *GPSFollower*, *GPSLeader*, *IRFollower*, *IRLeader*, and *ObstacleAvoider*. The *RoleNegotiator* implements a distributed negotiation to assign a role (leader or follower) to all robots in the convoy. The negotiation protocol ensures that only one robot can be assigned the leader role. Assignment of a role is a precondition for using any of the following components. Only the leader may use waypoint following and only followers may use the other types of following. *GPSFollower* and *IRFollower* require transmissions from a *GPSLeader* and *IRLeader* in the followed robot, respectively. *GPSLeader* transmits coordinates over the wireless LAN, while *IRLeader* emits an IR signal; these components are pulled into the target architecture as required interfaces of their respective follower components. As long as the robots are successfully following waypoints or another robot, they set the fluent $IsFollowing = true$. Whenever a robot gets stuck behind an obstacle, $BehindAnObstacle = true$, and the *ObstacleAvoider* is invoked, which implements an algorithm to move around an obstacle and bring the leader back into view. In addition to the above components, seven other sensor and actuator components are specified in SADEL. The interfaces of these components are required to read sensor values, steer the robot wheels, etc.

We specified an NPDDL problem description in which the goal is $(IsFollowing \wedge \neg BehindAnObstacle)$. The PLASMA planning layer (recall Figure 3), which is deployed on to a laptop, generates an NPDDL application domain model, application plan, adaptation domain model, adaptation goal (target architecture), and adaptation plan. The planning layer also automatically generates and compiles implementation code for an *Adaptation Analyzer* and *Executor*. The PLASMA planning layer then deploys compiled binaries of all required components (application components provided by the architect, *Adaptation Analyzer*, *Admins*, *Collectors*, etc.) and instantiates an identical adaptation layer on each robot. The adaptation layer on each robot instantiates the application layer and the *Executor* begins execution of the application plan, in which the first step is role negotiation. The deployed architecture is depicted in Figure 9.

As long as all robots are successfully following, the application remains in the goal state and the *Executor* does nothing. When an obstacle is encountered, the application is moved out of its goal state and the *Executor* invokes the *ObstacleAvoider* component. Furthermore, basic types of adaptations are automatically handled by the *Executor*. For example, if a robot is using camera following, and the area becomes too dark, the *Executor* can use the application plan to automatically switch to GPS or IR following. Similarly, GPS following does not work indoors, while IR following does not work at large distances. The *Executor* automatically handles these situations according to the application plan. A snippet of the application plan is shown in Figure 10. This small part of the plan shows instances in which the robot is assigned a leader, an obstacle is avoided during waypoint following, and robots start camera or IR following. The complete plan included 790 state-actions and took 300ms to generate on a mid-range computer. The initial adaptation plan was generated in 1.59s and included 1353 state-actions and 15 components.

More powerful forms of adaptation require dynamic replanning. Consider the case where a new system requirement is introduced at run-time in which the robots are required to recharge their batteries using docking stations along the route. If any robot has battery power less than 10%, the convoy is required to temporarily halt while that robot leaves the convoy, docks at the station, charges, and returns. To satisfy this requirement, we specified new SADEL models for a *BatteryMonitor* and *StationDocker*. Adding these SADEL models evolves the NPDDL application domain model description. At this point, the architect can either initiate replanning or let the current plans continue to run. However, the current goal of the system does not imply that charging should be performed, so replanning does not change the application. Therefore, we also specified a new application goal, $(IsFollowing \wedge \neg BehindAnObstacle \wedge \neg BatteryIsLow)$, and then initiated replanning. PLASMA computed new plans and regenerated and redeployed the *Adaptation Analyzer* and *Executor* components, along with the other required application components. Extending the requirements of the application in this way increased the number of state-actions in the application plan to 2318 and increased the generation time to 1.2s. The adaptation plan contained 4390 different state-actions and took 5.89s to generate.

Different types of failures can also cause adaptation and/or replanning within the application and adaptation layers, depending on the type and severity of each failure. For example, if the *ObstacleAvoider* component unexpectedly terminates due to a transient software bug, the current architecture of the application layer changes and is no longer equivalent to the target architecture. The *Collector* in the adaptation layer detects this and notifies the *Adaptation Analyzer*, which directs the *Admin* to perform an *Instantiate(ObstacleAvoider)* action according to the adaptation plan. Thus, this particular failure results in automatic adaptation without replanning. However, if the camera on one of the robots fails, the *CameraFollower* cannot be reinstantiated. In this case, the NPDDL application domain model is regenerated without the *CameraFollower* component, resulting in new application and adaptation plans that do not utilize the camera. The new application plan for this scenario was generated in 400ms and included 856 state-actions; the corresponding adaptation plan had 3110 state-actions and was generated in 3.74s.

4.2 Discussion

Our case study demonstrates how PLASMA-enabled applications transparently adapt to both *foreseeable* and *unforeseeable* conditions and requirements. Foreseeable conditions requiring adaptation, such as the failure of an unreliable component or the ability to perform object following in the dark, are built into application and adaptation plans. Consequently, adaptations that handle these conditions are automatically realized without incurring the performance penalties of replanning. Unforeseeable conditions and requirements, such as an unexpected type of failure or the need to recharge, are addressed through replanning directed by the system architect. Replanning incurs a performance penalty, but the situations where it is required are relatively rare, and the alternative – shutting down the application for an upgrade, for example – may be worse. The replanning times of <6s in all cases (and <1s in some cases) were well within the acceptable bounds for our case study. The ability to handle unforeseeable conditions and requirements in this manner sets PLASMA apart from similar approaches.

PLASMA is also designed to simplify the task of specifying when and how the system should adapt. Automatically generated plans in our case study ranged from 790 to 4390 state-actions. Manually specifying policies of this size would be tedious and cumbersome for the architect. To handle the goal change in the recharging example of our case study, for example, we only needed to specify two additional SADEL models (55 lines total) and a new NPDDL problem description (1 line of change), along with the implementations of the two new components. Moreover, the system architect maintains control over the process. The adaptations that result from new architect requirements are realized transparently, but the adaptations are only performed when the architect decides it is safe and appropriate to do so. Other types of adaptations, such as foreseeable adaptations and failures, are handled without architect intervention.

The use of planning within PLASMA has a number of natural benefits, which can be seen in the case study. First, a policy-based approach can result in policies that conflict with each other or collectively make it impossible for the system to achieve its goal. Both of these situations cannot arise in PLASMA's planning-based approach: if system constraints make the goal unachievable, PLASMA automatically notifies the architect before any adaptations are performed. Second, the adaptation plans generated in PLASMA preserve all software adaptation restrictions and requirements specified in the adaptation domain model. These requirements restrict architectural adaptation based on platform or application constraints. For example, the adaptation plan in our case study removes the *CameraFollower* after disconnection from the *Executer* in order to avoid a dangling port in the *Executer*, which can lead to communication failures.

5. RELATED WORK

Oreizy et al. introduced the concept of runtime software adaptation for architecture-based systems in 1998 [16]. In their follow-up work [17, 22], they reviewed and examined a larger number of adaptation approaches suggested in the decade hence. In this section we restrict ourselves to architectural approaches that rely on plans or policies. Readers are encouraged to refer to [17, 22] for more related work.

Among the policy-based approaches to self-adaptive systems, the most closely related approaches are PBAAM [9] and AURA [7]. AURA is a task-oriented (as opposed to PLASMA's goal-oriented) self-adaptive approach that considers qualities of service as primary factors to determine when to adapt a system. Besides using manually-specified policies rather than automatically-generated plans, AURA does not utilize ADLs. Georgas et al. [8, 9] introduced the PBAAM approach to self-adaptive systems. PBAAM is unique in its ability to dynamically evolve policies during runtime. However, the user is still required to manually specify architectural adaptations, which places more burden on the architect as compared to the automatically generated plans by PLASMA.

Salehie et al. [18] provide a taxonomy for self-adaptation and discuss opportunities for novel research in that area. They point out that few approaches utilize AI planning techniques for self-adaptive systems. Srivastava and Kambhampati [20] discuss the use of planning techniques for installing and running applications autonomously. However, their suggested use of planning for self-adaptive systems does not take an architectural perspective (specifically, their work does not cover addition, removal, or replacement of components or connectors). Arshad et al. [1] provide a planning-based framework for failure recovery in distributed systems. Although PLASMA also aids in failure recovery, Arshad et al.'s technique utilizes a dependency model to determine the extent to which a failure may propagate through a system. However, their framework requires the architect to directly work with domain models, while PLASMA extracts domain models from ADL models. Furthermore, dynamic addition of new components or services is not supported by their technique. In a follow-up approach, Arshad et al. [2] provide a framework called Planit for deployment and reconfiguration of distributed systems using AI planning. Similar to PLASMA, Planit takes an architectural perspective by using components and connectors for deployment and reconfiguration. However, Planit only handles starting, stopping, and connecting of components and connectors, but not their addition, removal, and replacement. Furthermore, like their previous approach, Planit models components and connectors directly in domain model descriptions.

The most closely related approach to PLASMA is the approach of Sykes et al. in [21]. Both Sykes' and our approaches are capable of selecting an appropriate topology for the software system's architecture during runtime from an ADL model. PLASMA and Sykes' approach also both use planning-as-model-checking techniques. PLASMA differs from Sykes' approach in several key ways, however. First, their proposed system is not capable of replanning in the case of a domain description change or system goal change. Second, their approach generated plans for application-specific functionality, but those plans cannot handle replacement, addition, or removal of components. Third, Sykes' approach utilized pre-programmed adaptation plans, while our approach generates such plans on-the-fly. Fourth, Sykes' approach and PLASMA differ in their utilization of domain models: domain models must be specified in Sykes' approach, while PLASMA generates these models from ADL models. Consequently, the architect does not directly deal with domain models in PLASMA.

6. CONCLUSION AND FUTURE WORK

In this paper, we introduced an approach to software adaptation that utilizes modeling and planning techniques in a meta-layered architecture for self-adaptation. Our approach simplifies the specification and use of adaptation mechanisms for system architects by freeing them from having to design the application architecture topology and plan for specific adaptations. As a result, the architect avoids the difficulty of designing plans for unforeseeable conditions such as changing requirements and runtime failures.

One of the benefits provided by MBP is planning for temporally extended goals. These kind of goals not only specify the desired final state of the domain but also specify the conditions which should hold during the execution of the plan. In the future, we will study and leverage this property to address non-functional requirements and quality properties more rigorously in PLASMA. In addition, we will investigate how to leverage contingent planning techniques [19] to reduce the frequency of replanning.

7. ACKNOWLEDGMENTS

The authors would like to thank the Test Resource Management Center (TRMC) Test and Evaluation / Science and Technology (T&E/S&T) Program for their support. This material is based in part upon work supported by the Department of Defense, United States Army, White Sands Missile Range, NM under Contract No. W9124Q-09-P-0230. This work has also been supported by the National Science Foundation under grant numbers 0312780 and 0820170, and by Bosch Research and Technology Centers.

8. REFERENCES

[1] N. Arshad et al. A Planning Based Approach to Failure Recovery in Distributed Systems. In *1st ACM SIGSOFT Workshop on Self-managed Systems*, 2004.

[2] N. Arshad et al. Deployment and Dynamic Reconfiguration Planning for Distributed Software Systems. *Software Quality Journal*, 15(3):265–281, 2007.

[3] P. Bertoli et al. MBP: a model based planner. In *Workshop on Planning under Uncertainty and Incomplete Information*, 2001.

[4] R. Eaton et al. Autonomous Farming: Modeling and Control of Agricultural Machinery in a Unified Framework. In *Mechatronics and Machine Vision in Practice*, 2008.

[5] G. Edwards et al. Architecture-driven Self-adaptation and Self-management in Robotics Systems. In *Int. Workshop on Software Engineering for Adaptive and Self-managing Systems*, 2009.

[6] D. Garlan et al. Rainbow: Architecture-based Self-adaptation with Reusable Infrastructure. *Computer*, pages 46–54, 2004.

[7] D. Garlan et al. Task-based Self-adaptation. In *ACM SIGSOFT Workshop on Self-managed Systems*, 2004.

[8] J. C. Georgas and R. N. Taylor. Towards a knowledge-based approach to architectural adaptation management. In *1st ACM SIGSOFT Workshop on Self-managed Systems*, 2004.

[9] J. C. Georgas and R. N. Taylor. Policy-based self-adaptive architectures: A feasibility study in the robotics domain. In *Int. Workshop on Software Engineering for Adaptive and Self-managing Systems*, 2008.

[10] F. Giunchiglia et al. Planning as Model Checking. In *5th European Conference on Planning: Recent Advances in AI Planning*, 1999.

[11] C. Henke et al. Advanced Convoy Control Strategy for Autonomously Driven Railway Vehicles. In *IEEE Conf. on Intelligent Transportation Systems*, 2006.

[12] J. Kramer and J. Magee. Self-managed Systems: an Architectural Challenge. In *Future of Software Engineering*, pages 259–268. IEEE Computer Society, 2007.

[13] S. Malek et al. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE Trans. Softw. Eng.*, 31(3):256–272, 2005.

[14] S. Malek et al. An architecture-driven software mobility framework. *Journal of Systems and Software*, In Press, Corrected Proof, 2009.

[15] N. Medvidovic et al. A language and environment for architecture-based software development and evolution. In *21st Int. Conf. on Software Engineering*, 1999.

[16] P. Oreizy et al. Architecture-based runtime software evolution. In *20th Int. Conf. on Software Engineering*, 1998.

[17] P. Oreizy et al. Runtime software adaptation: framework, approaches, and styles. In *Companion of the 30th Int. Conf. on Software Engineering*, pages 899–910. ACM, 2008.

[18] M. Salehie et al. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):1–42, 2009.

[19] D. Shaparau et al. Contingent planning with goal preferences. In *21st National Conf. on Artificial Intelligence*, pages 927–934. AAAI Press, 2006.

[20] B. Srivastava and S. Kambhampati. The Case for Automated Planning in Autonomic Computing. In *IEEE International Conf. on Autonomic Computing*, 2005.

[21] D. Sykes et al. From Goals to Components: A Combined Approach to Self-management. In *Int. Workshop on Software Engineering for Adaptive and Self-managing Systems*, 2008.

[22] R. N. Taylor et al. Architectural styles for runtime software adaptation. In *WICSA/ECSA*, pages 171–180, 2009.

[23] R. N. Taylor et al. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[24] P. Wurman et al. Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses. *AI Magazine*, 29(1):9–20, 2008.