Towards Self-Managed Executable Petri Nets

Klaus Marius Hansen and Weishan Zhang and Mads Ingstrup
Department of Computer Science
University of Aarhus
{marius,zhangws,ingstrup}@daimi.au.dk

Abstract

An issue in self-managed systems is that different abstractions and programming models are used on different architectural layers, leading to systems that are harder to build and understand. To alleviate this, we introduce a self-management approach which combines high-level Petri Nets with the capability of distributed communication among nets. Organized in a three-layer Goal Management, Change Management, and Component Control architecture this allows for self-management in distributed systems. We validate the approach through the Flamenco/CPN middleware that allows for self-management of service-oriented pervasive computing systems through the runtime interpretation of Coloured Petri Nets. The current work focuses on the Change Management and Component Control layers.

1 Introduction

A self-managing, or autonomic, system is one in which technology is deployed specifically for the purpose of managing other technology [2]. Parashar and Hariri [17] describe four types of challenges for research in this area: Conceptual challenges concerning how we understand autonomic systems, including models and abstractions of them; the Architectural challenges of what architecture can enable self-management at various levels of granularity, locally or globally and so they can be specified, implemented and controlled in a predictable and robust manner; Middleware challenges about what core services are needed to support realization of autonomic systems subject to particular and perhaps varying quality requirements; and finally Application challenges that are concerned with the programming, development and maintenance of autonomic applications.

Our work addresses middleware and application challenges in the context of self-managed pervasive computing systems. Pervasive computing systems [19] are characterized by inherent dynamism, context awareness, heterogeneity, and open-endedness. All of this stresses the need for self-management and for solutions that scale across heterogeneous platforms. A central component in the Hydra middleware [9] of which this work is part, is the Flamenco subsystem that is responsible for self-management in a service-based pervasive computing context. This paper introduces a high-level Petri Net [10] variant, Flamenco/CPN, of this subsystem. A previous paper introduces a complementary Semantic Web-based version [22].

The rest of this paper is structured as follows: first Section 2 discusses related work from the perspective of Parashar and Hariri. Next, Section 3 presents three basic scenarios of self-monitoring and self-management that we used to design, implement, and evaluate Flamenco. The design of Flamenco/CPN is presented in Section 4, its implementation in Section 5, and its evaluation in Section 6.2. Finally, Section 7 conclude.

2 Related Work

There have been at least two main conceptual approaches to building self-managed systems. One is inspired by traditional Artificial Intelligence with the explicit representation and interpretation of plans as a basis for action, e.g. as in the three-layer architecture by Kramer and Magee [12]. The other major conceptual approach is to build systems without any explicitly represented overall plan, e.g. inspired by the decentralized control in ant colonies [13], or by evolution as an adaptation mechanism [7]. In principle our approach of communicating Petri Nets can be used for realizing both conceptual approaches, but we specifically use the Kramer and Magee model to structure self-management.

It is an open topic how well these two approaches can be unified in a given system architecture, but arguably the human nervous system that provide some of the inspiration for the vision of autonomic computing encompasses both high level cognitive explicit/conscious planning as well as relying on lower level more emergent properties for self management and healing.

From an architectural point of view, Kramer’s and Magee’s [12] recently proposed reference model for self-
managed systems is based on an interpretation of Gat’s three layer architecture for autonomous robotics [6]. This model for self-management of software systems contrasts earlier approaches based on Sense-Plan-Act architectures [6] which have also been used in self-management systems (such as by [5]).

Kramer and Magee argue that handling self-management on an architectural level is appropriate in terms of level of abstraction and generality and casts Gat’s three layer architecture in terms of conceptual layers of a self-management system:

**Component Control Layer.** This layer includes sensors, actuators, and simple control loops. In a self-managed system, this layer consists of elements that perform application functions (“control loops”), reporting of state to upper layers (“sensors”) and facilities for creating, changing, and deleting elements (“actuators”)

**Change Management Layer.** Based on state reported from the Component Control Layer, the Change Management Layer executes precomputed plans and change the elements in the Component Control Layer. If a conditions is met for which a plan does not exist, a new plan may be requested from the Goal Management Layer.

**Goal Management Layer.** This layer creates new plans based on high-level objectives of the running system. Often compute-intensive re-planning is done. Although planning has been extensively researched in AI the application of these techniques to self-management is to the best of our knowledge unexplored.

Figure 1 illustrates the reference model.

In their seminal paper on autonomic computing, Garnek and Corbi [4] describe five degrees of autonomy. In this model, increasing the level of autonomy corresponds to requiring higher-level functionality in Kramer and Magee’s reference model. Since any system is unlikely to be completely autonomous there will always be certain high-level tasks a human operator must perform. Flamenco/CPN is specifically built in correspondence with the Kramer and Magee model.

From a middleware perspective we are concerned with what core services are needed to support the realization of self-managed computing systems subject to particular and perhaps varying quality constraints. Furthermore, support for distributed communication is also important here. We build on pervasive web services with embedded state machines [9] and with event-based communication using the publish/subscribe paradigm [3].

Finally, the application perspective is concerned with the programming, development and maintenance of autonomic systems. Barret et al. [1] performed ethnographic studies of systems administrators and emphasize the need for enabling awareness by operators, support them in rehearsal and planning activities and aid them in managing multitasking, interruptions and diversions. Similarly, we have performed ethnographic studies of agricultural, pervasive (maintenance) work [11] to derive our scenarios.

Concerning the programming of autonomic systems, [14] propose a component based framework in which each component is rule based and specified through behavior rules and interaction rules. Each component is itself self managing, a requirement also stated by White et al. [21]. We follow a similar approach in that both (device) web services and other subsystems may contain executable Petri Nets.

### 3 Scenarios of Self-Management

We base the description of self-management in Hydra on scenarios from the agricultural domain [11]. In the following, we outline a future scenario in which monitoring and management is involved.

Bjarne is an agricultural worker at a large pig farm. His daily routines include taking care of one of the slaughter pig stables, maintaining equipment, and helping with various jobs on the farm as needed.

To help him in his tasks, he carries around a PDA. Each morning, he feeds the pigs in one of the pig stables. When he enters the stable this morning, the PDA shows him that there is a leak in the water pipes somewhere, something that is critical for...
the well-being of the pigs. He quickly locates the problem and has his smith repair the leak.

Later the same day he notices a notification that says that he needs to take care of the Heating Ventilation and Air-Conditioning (HVAC) system. Luckily there is a standby system that has been turned on, otherwise his pigs would have suffered (and possibly died) within short time.

“Time for the big service check”, Bjarne mutters to himself as he sees the final notification of the day: one of the many thermometers at the farm is broken! Time to feed the pigs again...

Based on this, we derive the following scenarios of diagnosis and self-management:

### 3.1 Self-Management Scenario 1: Threshold

In the first part of the use scenario, Bjarne’s system is able to detect a broken water pipe based on measurements of characteristics of the flow meter of the system. This is a basic scenario where diagnosis of problems is based on simple threshold values of flow meter measurements. The diagnosis is based on the fact that in a working system, the water consumption of the pigs determine the flow level of the water in the system. The thresholds are:

- **Low.** If the flow meter values are too low, the pigs are either not drinking enough water or there is too little water coming into the farm. Both of these situations are problematic.

- **High.** If the flow meter values are high (but not extremely high), it may again be a sign of problems, e.g., due to too high temperature in the stable or sickness in the pigs.

- **Too high.** In this case, if the water flow is above this level, a water pipe is most probably broken.

This example of diagnosis is taken from the farm studied where such a system is in operation. Figure 2 illustrates the case. At time \( t_1 \), the water level is **Low**, at \( t_2 \) it is **High**, and at \( t_3 \) it is **Too high**.

### 3.2 Self-Management Scenario 2: Trend

In the HVAC part of the scenario, the diagnosis and repair is based on measurements from an indoor and an outdoor thermometer (see Figure 3) that is used to track the effects of the HVAC system. The diagnosis is more complicated, since a time trend of the two thermometers is needed to deduce the problem. There are two cases to consider.

More precisely, in case 2., the situation is problematic if \( \text{Tout} \) falls for a number of consecutive readings whereas \( \text{Tin} \) does not. In the figure, \( \text{Tout} \) starts falling at \( t_1 \) and \( \text{Tin} \) should have started falling at \( t_2 \), but it has not.

### 3.3 Self-Management Scenario 3: Interpretation

In the third self-management part of the use scenario, a thermometer stops working. This is detected through missing reports from the thermometer (“Status” in the Kramer and Magee model, cf. Figure 1). In this case, the user is notified about the problem which is not serious since a number of thermometers are still working. This self-management scenario illustrates the use of information from reflection on the system communication.
Concretely, this scenario assumes that Hydra is able to monitor network traffic between services and report and interpret that. Figure 4 shows the system model that the self-management of scenario 3 works upon when the network traffic has been interpreted.

![Figure 4. Self-Management Scenario 3: System Model](image)

In this chapter, we take this as an assumption that such a model can be created and in Chapter 4, we discuss the details of how it can be created from running Hydra systems. The “Host” class models a machine on a Hydra network. The Host has an IP address and may run a number of “Processes” each of which has a process ID (“pid”) \(^1\). Processes may communicate with other Processes (possible on other Hosts) by invocation of web services. In the model this is illustrated as an “Invocation” association class that also tracks the time of invocation. Furthermore, if a Process is a server for a web service, it will also have at least one open IP port.

In the concrete scenario, Hydra (and Flamenco) uses this information to deduce liveness of processes in the system. Figure 5 illustrates this. In the figure, the thickness of the Process objects (informally) signify the perceived aliveness of the Process. Here, Process 1 which is running on the Thermometer Host is believed to be dead. This information may now be used by a Hydra application.

These three scenarios will in the following be used to illustrate the design and implementation of first iteration of the Hydra self-management approach and also to evaluate the implementation.

### 3.4 Further scenarios of self-management

The three scenarios outlined above are all simple, yet shows several aspects of self-management. In this section we briefly outline the full use of self-management in Hydra:

- **Goal management.** Given a goal in a specific application, a Hydra-based should continuously optimize itself to support that goal. An example from the agriculture scenario would be that the response time from an alarm is generated somewhere until it is received by a user terminal should be less than a given threshold.

- **Self-healing.** Strongly connected to goal management as outlined above is self-healing in which an application attempts to be continuously running even in the event of partial failure.

- **Self-configuration.** A clear goal of much pervasive computing middleware is self-configuration in which, e.g., new devices and services announce themselves and existing services subsequently make use of these. This is supported, e.g., by UPnP. An additional layer on top of this would be that parameters of individual services should be continuously optimized based on the arrival (and departure of services).

- **Self-protection.** An example of self-protection in a self-managed system could be the ability to sense attacks and reacts upon those. Attacks could be sensed if traffic to and from the system is abnormal and a reaction to that could be to (temporarily) shut off communication to an application.

### 4 Design of Flamenco

In this chapter, we introduce and discuss the design of Flamenco. We base the design of Flamenco on the reference model of Kramer and Magee [12]. In doing so, we are currently focusing on the Change Management and Component Control layers. These are discussed next.

#### 4.1 Component Control

Hydra implements a service-oriented architecture based on web service interaction among devices. Thus a reason-
able granularity to build a self-management system on is the level of web service requests and responses. Furthermore, we are interested in the states of devices per se, i.e., is the device operational, stopped, not working and if it is operational what is the value of its sensor readings (if any) or its actuator state (if any).

This leads us to focus on status reporting of the following two forms:

- **State change reporting.** The Limbo web service generation tool [9] supports the generation of state machines (a restricted form of Petri Nets) describing the states of a device and its associated services. These state machines report their state changes as events through a publish/subscribe subsystem of Hydra. Furthermore, state machines may be specified in a way so that data from device services are sent as part of the events. An example of this would be a thermometer that has a *measuring* state that it continuously returns to after each physical measurement (see Figure 6); here the event sent may naturally contain the latest temperature measurement.

![Figure 6. A thermometer state machine](image)

- **Web service request/reply reporting.** The interaction among devices and managers in Hydra takes place via web service calls. These can then be said to correspond to “system events” in the sense of Schmerl et al. [20]. The requests and replies (and their associated data) can subsequently be used to analyze the runtime structure of running Hydra systems. To do this, we have implemented an IP sniffer that is able to report about TCP/IP packets being sent between hosts. The sniffer currently runs on Windows only. For other platforms, the Limbo compiler is able to produce adapters that report system events.

Currently, we assume that the management interface of device web services is realized through their subscription to change events, but a more refined management interface needs to be designed.

### 4.2 Change Management

Following up on Kramer and Magee [12], we approach self-management through *architectural abstractions*. Thus an integral part of Flamenco then becomes to make sense of system level events (“Status”) and transform that into representations of, e.g., architectural components and connectors.

Schmerl et al. [20] presents a recent approach to this that uses a domain-specific language and a formal runtime semantics of this language described using Coloured Petri Nets (CP Nets) [10]. Flamenco/CPN generalizes this to use a state-based approach, more specifically a Petri Net-based approach, to do interpretation of system events as architectural events and to reason upon these elements.

In the following, we first briefly present CP Nets and how we use this formalism in Flamenco/CPN.

#### 4.2.1 Coloured Petri Nets

Coloured Petri Nets is a formal, graphical modeling language with well-defined syntax and semantics. Here, we provide a very brief and somewhat informal introduction to CP-nets which is adapted from [10] and [8]. The structure of a non-hierarchical CP-net is formally defined as a tuple:

**Definition 1 (Coloured Petri Net)** A non-hierarchical CP-net is a tuple \( CPN = (\Sigma, P, T, A, N, C, G, E, I) \), where

- \( \Sigma \) is a finite set of non-empty types called colour sets;
- \( P, T, \) and \( A \) are non-empty finite, disjoint sets of places, transitions, and arcs, respectively;
- \( N \) is a node function defined from \( P \times T \) into \((P \times T) \cup (T \times P)\);
- \( C \) is a colour function defined from \( P \) into \( \Sigma \);
- \( G \) is a guard function defined from \( T \) into boolean expressions;
- \( E \) is an arc expression function defined from \( A \) into boolean expressions such that the arc expression for an arc evaluates to a multi-set of values from \( C(p) \) where \( p \) is the place that the arc is connected to; and
- \( I \) is an initialization function defined from \( P \) into boolean expressions that do not contain variables such that the initialization expression for place \( p \) evaluates to a multi-set of values from \( C(p) \).

#### 4.2.2 Integrating CP-nets and Publish/Subscribe systems

An integral part of the Hydra architecture is a publish/subscribe [3] subsystem. The Limbo compiler, e.g., generates services for devices that advertise their state through notifications. A similar pattern is used in other kinds of pervasive computing middleware such as...
Definition 2 A publish/subscribe system is a tuple $PS = (C, N, F)$, where

- $C$ is a non-empty, finite set of clients of the system;
- $N$ is a set of notifications; and
- $F$ is a filter function from $N$ into boolean expression

We now informally describe the runtime semantics of such systems. An execution of a publish/subscribe system is a trace of states of the system and operations on the system. For a formal definition, refer to [16]. The state of the system defines for each client in $C$ which filters it has in place (what it has subscribed to) and which publications each client has made. The operations are on the form:

- $subscribe(c, f), c \in C, f \in F$ resulting in $c$ subscribing to notifications that evaluate to true when $f$ is applied to them
- $unsubscribe(c, f), c \in C, f \in F$ resulting in that $c$ does not subscribe to $f$
- $notify(c, n), c \in C, n \in N$ where $c$ is notified about notification $n$. In a safe system, only clients that are subscribed to a filter that applied to $n$ evaluates to true should be notified of $n$
- $publish(c, n), c \in C, n \in N$ meaning that $c$ publishes a notification $n$. Eventually, clients, $c'$ for which $\exists f \in F : subscribe(c', f) \land f(n) = true$ are notified of $n$

An example trace for a publish/subscribe system, $ps$, could be

$$\sigma = s_0, subscribe(c_1, f_1), s_1, publish(c_2, n_1), s_2, notify(c_1, n_1), s_3, ...$$

Here $c_1$ subscribes to $f_1$, $c_2$ publishes a notification $n_1$ (where $f_1(n_1) = true$) and subsequently $c_1$ is notified of $n_1$. In the trace, $s_0, s_1$ etc. are the states of the publish/subscribe system.

Now, to combine publish/subscribe systems and CP-nets in Flamenco/CPN, we require that certain notifications in an execution trace of a publish/subscribe system are mapped to tokens in an execution of a CP-net. More precisely, we define:

**Definition 3** A Flamenco system is a tuple $(F_{PS}, F_{CPN}, p_{in}, p_{out}, F_{in}, m_{in}, c_F)$, where $F_{PS} = (C_{PS}, N_{PS}, F_{PS})$ is a publish/subscribe system and $F_{CPN} = (S_{CPN}, T_{CPN}, A_{CPN}, N_{CPN}, C_{CPN}, G_{CPN}, E_{CPN}, I_{CPN})$ is a CP-net, and where:

- $p_{in} \in P_{CPN}$ is an input place;
- $p_{out} \in P_{CPN}$ is an output place, $p_{out} \neq p_{in}$;
- $F_{in} \subseteq F_{PS}$ is a set of input filters; and
- $m_{in}$ is an invertible function from $N_{PS}$ to $C_{CPN}(p_{in})$
- $c_F \in C_{PS}$ is Flamenco client

The basic idea now is that whenever a notification is published that matches a filter in $F_{in}$, the notification is “converted” to a token in the executing CP-net that is placed on $p_{in}$. Conversely, tokens in the CP-net execution that are placed on $p_{out}$ should be “converted” to a notification. More precisely, we want the following two properties to hold:

1. For all traces, $\sigma$, of $F_{PS}$, if $publish(c, n), c \in C_{PS}, n \in N_{PS} \land \exists f \in F_{in} : f(n) = true$ and the marking of $F_{CPN}$ is $M$ then there will be a subsequent marking, $M'$, so that $m_{in}(n) \in M'(p_{in})$
2. For all markings, $M$, of an execution of $F_{CPN}$, if $\exists t \in M(p_{out})$ then the trace of $F_{PS}$ will contain $publish(c_F, m_{in}^{-1}(t))$ and there will be a subsequent marking, $M'$, so that $t \notin M'(p_{out})$

A simple example

Figure 7 shows a very simple example of a Flamenco CP-net. The example is cast in the context of Hydra in which the publish/subscribe system is topic-based, notifications consist of topics and events, and events are untyped (actually string-based) key-value pairs. Filtering is straightforward in this case: clients are notified if they subscribe to the topic (or a super-topic of a topic) that another client publishes on. In Figure 7, the input place is the “Input Events” place with colour “INPUT” and the output place is the “Output Events” place with colour “OUTPUT”. In the example, all publications to the topic “/statemachine/statechange” will be converted to a token on “Input Events”. Whenever this happens, a token will be placed on “Output Events” and this token will eventually be converted to a notification on the topic “/tick”. As a side effect, “Event Count” contains a token that counts the number of notifications that have been received on the topic “/statemachine/statechange”. The count is also used in the event that is part of the “/tick” notification.
5 Flamenco/CPN Implementation

Flamenco/CPN is realized in a distributed fashion in the Hydra middleware. With respect to qualities, the architecture should support building a system that is adaptable and where devices are easily installable. This implies, e.g., that there should be a loose coupling between Flamenco/CPN and the rest of an application. Furthermore, efficiency is certainly an issue: there should be little impact of devices in terms of time behavior and resource utilization.

This is to a certain extent achieved by coupling Flamenco/CPN to the publish/subscribe subsystem of Hydra (i.e., the Event Manager) and taking advantage of that Limbo-generated services already use the Event Manager to publish state information. Further work also involves making the execution itself of the Flamenco CP-net distributed.

Figure 8 shows a runtime view of the Flamenco/CPN architecture. Here a set of Devices publish their state using the PublishSubscribe component. The Flamenco/CPN component is responsible for receiving these notifications and convert them to input CP-net tokens and for consuming output CP-net tokens when available. Flamenco/CPN does this by interacting with CPN Tools [18]. CPN Tools is responsible for the Flamenco CP-net including for the execution of it. Figure 9 shows a dynamic view of these components interacting in the form of a sequence diagram.

6 Evaluation of Flamenco/CPN

This section presents two types of evaluation of Flamenco/CPN:

- A qualitative evaluation in which the three scenarios from Section 3, are realized using Flamenco/CPN
- A quantitative evaluation of performance and scalability of Flamenco/CPN based on the first self-management scenario (Section 3.1)

6.1 Qualitative Evaluation

We realized the three self-management scenarios. In general, the realization of the CP-nets were straightforward in the sense that modeling was unproblematic. Work is still needed to enhance support for the programming model since CPN Tools has to be used directly and in a centralized way.

Figure 10 shows the CP-net for the first self-management scenario. The net follows a simple structure in which the three thresholds are realized as parameters (“LOW”, “HIGH”, and “TOO HIGH”). These parameters are compared with state of input from flow meters in the transitions “Low Flow”, “High Flow”, and “Too High Flow” in which the parameters and the flow meter value are used in the guards. If one of these transition fire, an alarm (on the topic “/diagnosis/alarm”) is generated.
In figure 11 we show the CP-net that implements the part of scenario 3 that pertains to constructing an architectural model of the Hydra middleware. In this case, the “Hosts” place contains a list of products of type (host, processid). The “Invocations” place contains a list of invocations of type (localhost, remotesite) that indicate that a process on host localhost has invocation a process on host remotesite. The “Get Invocations” transition’s guard finds invocations by assuming that two partial invocation tuples from the Flamenco sniffer belong to the same invocation (is a request and reply) if they match in hosts and ports and if the are sufficiently close in time. We did not implement the diagnosis part of the scenario since this has been evaluated in the previous scenarios.

6.2 Quantitative Evaluation

We did a quantitative evaluation of Flamenco/CPN based on Scenario 1 from above.

The measurements were made on MacBook Pro with a 2.4 GHz Intel Core 2 Duo processor, 4 GB 667 MHz DDR2 SDRAM, and Mac OS X Version 10.5.1. Since CPN Tools only run on Linux and Windows, we ran Flamenco/CPN in VMWare Fusion Version 1.1. The tests were run only on a single machine since it was not our intention to measure network performance. Clearly, for a distributed system such as Flamenco it is of interest to measure the network-related performance, but in this case we were interested in the performance of the actual reasoning.

In the test setup, a Tester (implemented using Limbo) continually publishes state changes of a flow meter using the Event Manager. A Semaphore controls how many concurrent publications the Tester makes. If, e.g., the Semaphore has three permits, three concurrent publications can be made before the tester blocks on the Semaphore. For each publication by the Tester, Flamenco/CPN should publish an alarm event.

When a response is received from Flamenco/CPN (via the EventManager), a permit of the Semaphore is released and another publication can be made. This is used to emulate a number of devices publishing and to used to measure scalability (see Section 6.2.2). This effectively gives two parameters to vary for the evaluation:

- **Number of devices** which is controlled by number of permits of the Semaphore
- **Number of publications** which is controlled as a parameter to the Tester.

The tester publishes the total number of publications as quickly as possibly, only blocking as an effect of acquiring the Semaphore.

6.2.1 Performance

For the performance measurements, we ran the Tester with 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 500, and 1000 publication and one permit in the Semaphore. We ran each sub-scenario four times and took the average of the execution times of the last three (to allow Flamenco/CPN to start up and stabilize). The average execution time was 32.2 msec, the median was 31.0 msec and the variance was 10.2. This execution time is for a “round-trip”including execution in CPN Tools.

As Figure 12 shows, Flamenco/CPN appears to scale linearly in number of publications.
6.2.2 Scalability

For the scalability measurements, we simulated 1, 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100 devices each of which published 10 events. The average execution time was 29.4 msec, the median 28.5 msec, and the variance was 9.1. It can be observed that execution times are lower in average than in the performance measurements. This is most probably due to the fact that the EventManager is multithreaded and that this is exploited in the scalability tests. Furthermore, it can be noticed that the Event Manager is a bottleneck in this evaluation, preventing a real scalability evaluation. A second set of experiments where the tester created a number of concurrent threads that published state change data were created. Here Flamenco/CPN behaved well until 80 threads were running simultaneously after which Flamenco/CPN failed.

Again, cf. Figure 13, Flamenco/CPN in the test scales linearly.

7 Conclusions

This paper has introduce Flamenco/CPN, a tool for self-management in service-based pervasive computing. The tool is an instance of a general approach in which systems are modeled and implemented as communicating, high-level Petri Nets. To scale the nets to embedded devices, a restricted version of Petri Nets (essentially state machines) are used. In the reasoning part of Flamenco/CPN, full Coloured Petri Nets are used. Furthermore, Flamenco/CPN is realized according to the three-layer architecture for self-managed systems described by Kramer and Magee.

We designed and evaluated the tool in the context of agricultural scenarios, a domain with heterogeneous devices with complex interaction, exemplifying pervasive computing. Our evaluations in this context were in general encouraging: performance and scalability appears to be good and expressiveness (in terms of ability to realize the scenarios) is also good. However, further quantitative measurements are needed in particular with respect to scalability. It will, e.g., be interesting to know how the size and complexity of the Flamenco CP-nets affect performance. Moreover, memory consumption has not been measured.

Future work includes realizing a full Goal Management layer, work in which an integration with our Semantic Web-based version of Flamenco will be performed.

Acknowledgments

The research reported in this paper has been supported by the Hydra EU project (IST-2005-034891).

References


