Towards Multi-Models for Self-* Cyber-Physical Systems

Hansen Salim and John Fitzgerald

School of Computing, Newcastle University, UK
{h.salim,john.fitzgerald}@ncl.ac.uk

Abstract. The ability of systems to respond autonomously to environmental or internal change is an important facet of performance and dependability, particularly for cyber-physical systems in which networked computational and physical processes can potentially adapt to deliver continuity of service in the face of potential faults. The state of the art in the design of such systems suffers from limitations, both in terms of the capacity to model cyber and physical processes directly. We consider an approach to the modelling of cyber-physical systems that may reconfigure autonomously in accordance with internally stored policies. Specifically, we consider an adaptation of the MetaSelf framework to multi-paradigm models expressed using the INTO-CPS tool chain. We conduct a feasibility study using the benchmark UAV swarm example.

1 Introduction

The term *self-** ("self-star") is applied to systems in which humans define general principles or rules that govern some aspect of the system, but leave it up to the system itself to make the operational decisions within the bounds dictated by such policies. The class includes systems that are, for example, self-adapting, self-organising, self-healing, self-optimising, and self-configuring systems. Examples of specific applications include highly distributed swarm-like systems, with localised decision-making, or the more centralised architectures of reconfigurable manufacturing plants.

As in the examples above, many self-* systems are cyber-physical systems (CPSs), being composed of networked computational elements that often interact with physical processes have the capacity to adapt in response to external or internal events. Such adaptation may include the dynamic alteration of operating parameters in the cyber or physical elements, as well as modification at the architectural level through the dynamic incorporation of new components, or the establishment of new connections between existing components. Such mechanisms could, for example, provide consistent and enhanced resilience to unanticipated events. The challenge then arises of undertaking model-based engineering for self-* CPSs, which involves the selection of optimal policies governing operation, with the need to maintain assurance of dependability.

The MetaSelf framework [5] was proposed to support the description of dynamically reconfigurable systems, providing a basis for delivering assurance on some properties of self-* capabilities. Originally proposed as a means of supporting the architectural description and verification of dependable self-adaptive and self-organising systems, it has only been explored in a digital (cyber) context. In this paper, we begin to examine the feasibility of engineering self-* CPSs within the INTO-CPS co-modelling and co-simulation framework. We first introduce key concepts in self-* systems and identify related work (Section 2), and we then focus on the specific MetaSelf framework (Section 3). We propose an approach to realising MetaSelf in INTO-CPS (Section 4) and explore the feasibility of this approach by means of a case study based on the UAV swarm example (Section 5). We draw conclusions in (Section 6).

2 Background: Self-* Systems and Reconfiguration

The term "Self-*" refers to systems that exhibit some autonomy [2] and has been described as one of the defining characteristics of CPSs [1], as well as a key to future CPSs [18]. Research in CPSs with self-* properties has demonstrated benefits in areas such as scalability [10], functionality [3], resilience [14,17], and optimization [6]. Examples exist in many domains. For example, it has been shown that self-healing can enable a more reliable smart grid [7,9] in which the self-* mechanism uses real-time control over power sources and switches to reconfigure the distribution network to isolate faults. A Generic Adaptation Mechanism (GAM) has been proposed for the ICT architecture of smart cars [15]. In robotics, modular self-reconfigurable units enable variable morphology [19] by rearranging their connectivity to adapt to unexpected situations, perform new objectives, or recover from faults.

Designing a self-* enabled CPS is far from trivial, especially when considering the interoperability of inherently heterogeneous CPSs. CPS designers must also consider the difference in dynamics between cyber and physical elements, as well as ensuring that autonomy can be achieved within time bounds.

One favourable approach is the use of Service-Oriented Architecture (SOA) to co-ordinate computational and physical processes. In an SOA services are provided by agents/components accompanied by a middleware framework that unifies the network [8,12,16]. SOA appears promising, but research on its application to self-* CPS is still lacking. Current approaches are described for specific domains: it is not yet clear that results are readily transferred to other CPS domains. Leitao suggests a 3-layer framework as for the engineering of adaptive complex CPSs [11], involving the integration of SOA with multi-agent systems for interoperability, using cloud computing and wireless sensor networks to enable ubiquitous environment, and self-* properties through self-organization

techniques. While the study shows the potential of the technology, the research is still at its infancy without (so far as we currently know) a proof of concept.

SOA has also been adopted by Dai et al. [4] to enable "Plug-and-play" in industrial CPSs with a case study on lighting systems. The "Plug-and-play" refers to the controller level software services that integrate loosely coupled automation systems through service discovery and orchestration. However, there is no support to describe this framework outside of industrial CPS.

3 MetaSelf

MetaSelf is an architectural framework that supports dynamically reconfigurable systems, providing a basis for gaining assurance in some self-* capabilities [5]. It exploits metadata and internally stored policies to enforce reconfiguration decisions. MetaSelf utilises SOA and, as mentioned in Section 2, the SOA approach has potential benefits in the design of CPS. MetaSelf was originally proposed as a framework for trustworthy self-adaptive and self-organising systems and in this section, we highlight the potential of adapting MetaSelf for CPSs, looking at how the characteristics of the MetaSelf architecture might be able to tackle certain CPS design challenges. We then describe the MetaSelf Development Process which will be used on our case study in Section 5.

3.1 MetaSelf Architecture

The main elements of MetaSelf include the following:

Self-Describing Components/Services/Agents. Component decoupled from their descriptions (e.g functionalities, policies, QoS) are a primary characteristic of SOA. This approach allows for interoperability as well as run-time planning for solutions to a priori unknown changes.

Acquired, Updated & Monitored Metadata. MetaSelf relies on acquiring, updating and monitoring metadata to enable self-* mechanisms. Metadata in MetaSelf can be related to individual or groups of components (e.g., QoS, availability), description of system components (relates to component interface), or the environment (relates to information received from system sensors). These metadata support decision-making for self-adaptation and/or self-organisation.

Self-* Mechanisms. Self-* mechanisms describe how and when reconfiguration is triggered. A mechanism takes metadata as a basis and adaptation actions are followed by all components. Self-* mechanisms can be implemented as *rules for self-organisation* or *dependability policies*. *Rules for*

self-organisation refer to the design of self-organisation that works bottom-up. Self-organising rules apply to each individual component and the resulting global behaviour emerges from their local interaction. *Dependability policies* refers to self-adaptive approach that works top-bottom, meaning that the run-time infrastructure evaluate its own global behaviour and change it when there is room for improvement.

Enforcement of Policies. The run-time infrastructure needs services that are able to enforce the policies as described by self-* mechanisms. These services may replace and reconfigure components and act directly on components.

Coordination/Adaptation. The list of components within the system is managed by this service. It connects and activates components to follow the automated reconfiguration actions following the rules set by the policies.

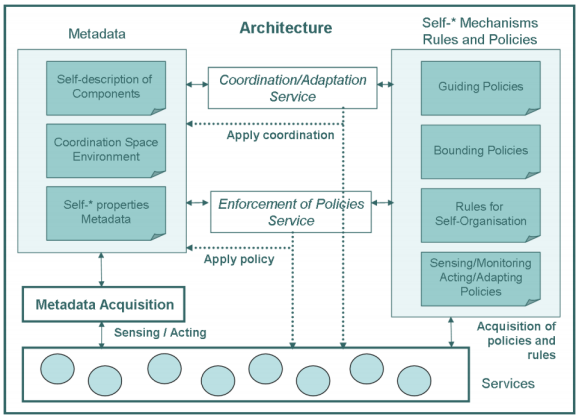

Fig. 1. MetaSelf run-time generic infrastructure

3.2 MetaSelf Development Process

The MetaSelf Development Process is a guide to designing the run-time infrastructure of MetaSelf-enabled systems. It includes four activities or phases . The *Requirement and analysis phase* describes system functionality and its self-* requirements. It should specify concrete trigger conditions of the desired reconfiguration and may include QoS. The *Design Phase D1* selects self-* mechanisms and architectural patterns along with the definition of generic rules/policies. The *Design Phase D2* includes description of policies and required metadata, component design, and any adaptation to the self-* mechanisms. The *Implementation phase* is the result of the run-time infrastructure based on the specified

design. A *Verification phase* covers activities intended to increase confidence that global properties and requirements are respected. Analysis and improvement may be identified and carried out in this phase.

4 Modelling MetaSelf-enabled Systems in INTO-CPS

INTO-CPS[1] is a tool chain that supports co-modelling of CPSs from requirements, through design, to realisation in hardware and software, enabling traceability at all stages. The INTO-CPS approach uses co-simulation based on the FMI standard, encapsulating models from different tools in FMUs, enabling inter-FMU communication and importing into hosting tools. The tool chain allows developers to build system models from diverse modelling tools and collaborate on multi-models.

As a first step towards integration of MetaSelf-enabled systems in INTO-CPS multi-models, we propose a stand-alone MetaSelf FMU that contains the main elements of the MetaSelf architecture. This FMU can be included as part of INTO-CPS multi-model design to enable reconfiguration and provide self-* characteristics. The MetaSelf FMU is a VDM-RT discrete-event (DE) model that requires metadata as its input, and outputs commands to other FMUs within the simulation for architectural reconfiguration purposes (Fig. 2). The MetaSelf FMU supports the realisation of MetaSelf architecture through VDM-RT files: `controller.vdmrt`, `component.vdmrt`, and `service.vdmrt`.

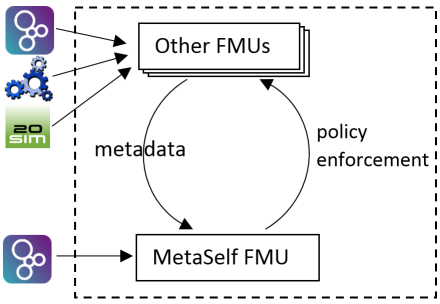

Fig. 2. Interaction between MetaSelf FMU with other FMUs in INTO-CPS multi-model

The file `component.vdmrt` defines a superclass that supports the description of generic components. The `Component` class contains methods to acquire its metadata and to enforce policies upon it that must be defined by its

[1] `http://projects.au.dk/into-cps/`

subclasses (See Fig. 3). This approach decouples the component from its description (i.e., metadata), supporting the *Self-Describing Components* element of MetaSelf.

```
class Component

operations
public getMetadata : token ==> real
getMetadata(t) == is subclass responsibility;

public enforcePolicies : Service ==> ()
enforcePolicies(c) == is subclass responsibility;

end Component
```

Fig. 3. Snippet of component class

```
system System
operations
public System : () ==> System
System () ==
 (
  --Attach connection to other FMUs to each corresponding component object
  let metadata1 =  new Sensor(hwi.metadata),
      enforcement1 = new Command(hwi.enforcement),
      myComponent1 = new Component(metadata1,enforcement1),

      metadata2 =  new Sensor(hwi.metadata),
      enforcement2 = new Command(hwi.enforcement),
      myComponent2 = new Component(metadata2,enforcement2)
  in
  (
    controller := new Controller(myComponent1, myComponent2);
    mainthread := new Thread(25, controller, myComponent1, myComponent2);
  );
  -- deploy the controller
  cpu.deploy(mainthread);
 );
end System
```

Fig. 4. FMU shared variables handled in `system.vdmrt`

The `service.vdmrt` defines a superclass that supports the description of metadata. It simply carries a token to identify the type of metadata it is describing. This superclass is especially useful when describing multiple type of metadata within a component.

The `controller.vdmrt` is where the main control loop will be handled. Metadata published by external FMUs will be acquired and saved as a local variable here. To support policy enforcement, local variables for component class objects are also defined and connected to the output port. *listOfComponents*

```
class Controller
instance variables
--SA/SO Adaptation
  public listOfComponents: set of Component;
  public registry : set of Component ;

operations
  -- constructor for Controller
  public Controller : Component * Component ==> Controller
  Controller (myComponent1, myComponent2) ==
  (
    --Component
    listOfComponents := {myComponent1,myComponent2};
    registry := {};
  );
```

Fig. 5. Snippet of controller.vdmrt (instance variable and constructor)

holds the active components within the system, each mapped with an identifier, while *registry* holds the non-active components (Fig. 5).

As described in Section 3.1, the pattern for a self-* mechanism is split into *rules for self-organisation* and *dependability policies*. A designer can define the desired self-* mechanism at *SOrules* and *dependabilityPolicies* method which will be executed at every simulation step (Fig. 6). Fig. 7 shows how each MetaSelf element is represented by the FMU.

5 A Feasibility Study: UAV Swarm

This section shows how the MetaSelf FMU approach has been applied to enhance the "Swarm of UAV" example from the INTO-CPS compendium [13] with self-* characteristics. The example concerns a group of Unmanned Aerial Vehicles (UAVs) that communicate to achieve global behaviour. Each UAV can adjust its pitch, yaw and roll to move in 3D space using rotors, and a central controller dictates the desired movements of the UAVs in the swarm.

5.1 Requirements and Analysis Phase

The goal of our swarm is to coordinate UAVs to provide wireless coverage over a designated area. UAVs serve as wireless routers and have fixed range of effective coverage radius. The UAVs should be coordinated to cover a rectangular area. For this case study, the area can be fully covered by aligning four UAVs (Fig. 8).

Each active UAV has a limited battery life and will have to leave the formation when this drops below a certain level. Our goal is to design a self-adaptive swarm using *dependability policies*. Requirements include:

```
-- POLICY
public Step : () ==> ()
Step() ==
(
  dependabilityPolicies();
  SOrules();
);

public dependabilityPolicies: () ==> ()
dependabilityPolicies() ==
(
  --Define the dependability policies here
  --E.g if components.metadata = faulty() then (replaceComponent();moveToRegistry();)
  return
);

public moveToRegistry : token ==> ()
moveToRegistry(t) ==
(
  registry := registry ++ {t |-> listOfComponents(t)};
  listOfComponents := {t} <-: listOfComponents;
);

public moveToComponentList : token ==> ()
moveToComponentList(t) ==
(
  listOfComponents := listOfComponents ++ {t |-> registry(t)};
  registry := {t} <-: registry;
);

public SOrules: () ==> ()
SOrules() == (
  --SOrules typically applies to all components and dictates its global behaviour
  --E.g if components.metadata = faulty() then replaceComponent()
  return
);
```

Fig. 6. Snippet of controller.vdmrt (Self-* mechanisms)

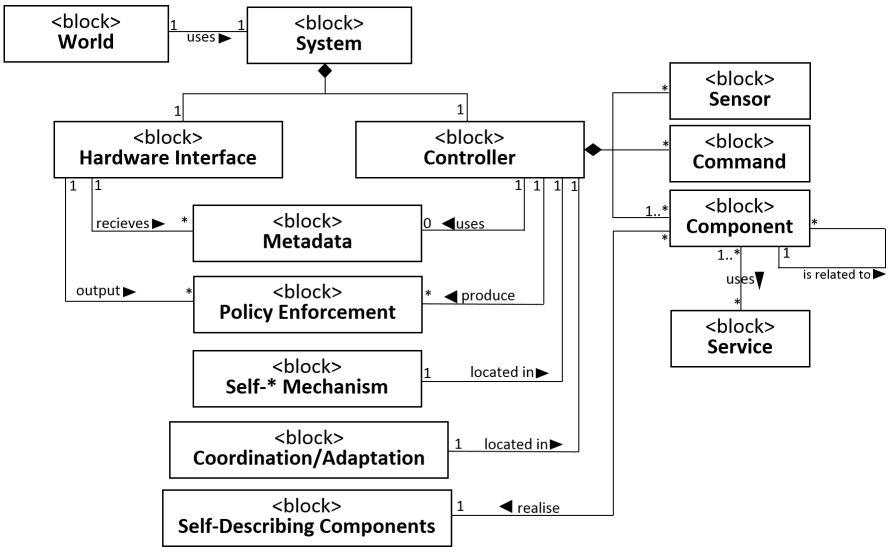

Fig. 7. MetaSelf elements in INTO-CPS FMU (SysML block definition diagram)

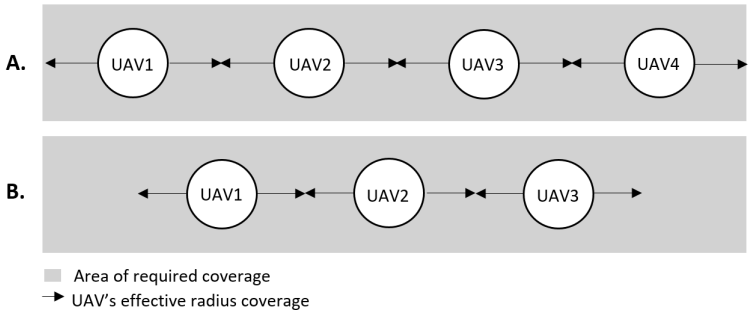

Fig. 8. UAV swarm. A: Optimum coverage with 4 UAVs. B: Example formation with limited UAVs.

1. The system could be equipped with fully charged backup UAVs. When there is a backup UAV, the system should be able to autonomously replace low battery UAV with the backup UAV.
2. When the number of available UAVs does not allow for optimum coverage (e.g., only 3 UAVs available), the system should be able to coordinate the UAVs to maximise the coverage, with highest priority being the center of the area.

For this study, we set up a scenario that starts with 5 fully charged UAVs and eventually with only 3 UAVs remaining at the end of the simulation.

5.2 Design and Implementation Phases

The multi-model has four FMUs. The MetaSelf FMU requires the battery level of each UAV as metadata, and will output coordinates to enforce the policies. In this phase, we design the SysML architecture diagram and connection diagram to show the architecture as well as the metadata flow of the swarm (Figs. 9 & 10).

To achieve the requirements fromSection 5.1, we need to decide how a self-* mechanism will be designed for the UAV swarm. The existing swarm model is made up of centrally controlled UAVs which are not equipped with sensors to monitor their environment, and therefore they are not able to make self-organising decisions. Consequently, the self-* mechanism design for this case study will follow a self-adapting pattern. Rules for self-adapting system (the *Dependability Policy*) are expressed in the `dependabilityPolicies` method. Metadata regarding the list of UAVs and battery life will be available to the policy, and it may call methods from *policy enforcement* to achieve self-adaptation. *Policy enforcement* has a method to activate (`moveToComponentList`) and

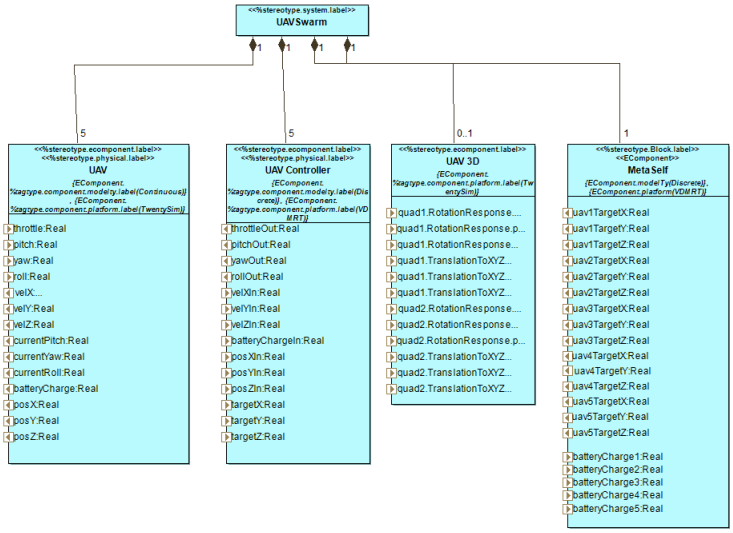

Fig. 9. SysML architecture diagram of UAV swarm

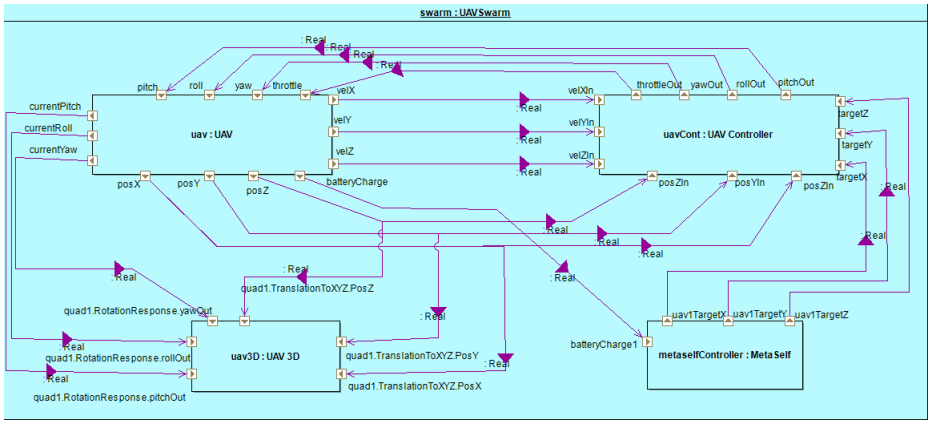

Fig. 10. SysML connection diagram of UAV swarm

deactivate (`moveToRegistry`) a component (Fig. 6) that will result in UAV replacement and method to set the UAV target coordinate (Fig. 11).

To represent the UAVs, we create a subclass UAV containing a method that provides metadata on battery life, and that may receive coordinates from the policy enforcement (Figs. 11). Figs. 12 and 13 refer to the controller class where *Dependability Policy* is described and enforced.

```
class UAV is subclass of Component

instance variables
public tarX: [Command];
public tarY: [Command];
public tarZ: [Command];
public battery: [Sensor];

operations
public UAV : Command * Command * Command * Sensor ==> UAV
UAV(x,y,z,batteryLife) == (
  tarX := x;
  tarY := y;
  tarZ := z;
  battery := batteryLife;
);

public getMetadata: token ==> real
getMetadata(t) == (
  if (t = mk_token("battery")) then return battery.GetValue() else return 0;
);

public enforcePolicies: Coor ==> ()
enforcePolicies(c) == (
  tarX.SetValue(c.GetX());
  tarY.SetValue(c.GetY());
  tarZ.SetValue(c.GetZ());
);
```

Fig. 11. Snippet of UAV class

```
class Controller
instance variables
  --UAVs will be set to fly along the following Y & Z axis
  private yCoor : real := 6.0;
  private zCoor : real := 4.0;

  --Center of coverage
  private xStart : real := 5.0;

  --The UAV's radius of effective coverage
  private radiusCoverage : real := 1.0;

  --Battery level benchmark for policy
  private minLevel : real := 6455;
  private normalLevel : real := 9000;

  --Max number of UAVs deployed & other variables
  private maxUAVdeployed : int := 4;
  private counter : int := 0;
  private flag : bool := false;
  private numOfUAVs : int;

--SA/SO Adaptation
  public listOfComponents: set of Component;
  public registry: set of Component;

--Components
  public uav1: [UAV] := nil;
  public uav2: [UAV] := nil;
  public uav3: [UAV] := nil;
  public uav4: [UAV] := nil;
  public uav5: [UAV] := nil;
```

Fig. 12. Variables of controller class

```
-- POLICY
public Step : () ==> ()
Step() ==
(
  dependabilityPolicies();
);

public dependabilityPolicies: () ==> ()
dependabilityPolicies() ==
(
  --Requirement 1: Remove low battery UAVs to registry
  for all component in set listOfComponents do (
    let batteryLevel = component.getMetadata(mk_token("battery")) in (
      if batteryLevel < minLevel then moveToRegistry(component);
      if batteryLevel > normalLevel then moveToComponentList(component);
    );
  );

  --Re-coordinate the UAVs when a UAV is replace/added
  if(flag) then (controlUAVs(); flag := false);
);

--Requirement 2: Maximize coverage when there is not enough UAVs
public controlUAVs: () ==> ()
controlUAVs() == (
  --Calculate how many UAVs can be deployed
  if (card(listOfComponents) < maxUAVdeployed) then
  numOfUAVs := card(listOfComponents) else numOfUAVs := maxUAVdeployed;

  --Calculate effective coordinate based on number of UAVs available
  let startPoint = xStart-(radiusCoverage*(numOfUAVs-1)) in (
    for all component in set listOfComponents do (
      if (counter < maxUAVdeployed) then (
        --Enforce new coordinates to the UAVs
        component.enforcePolicies(new Coor(startPoint+(radiusCoverage*2*counter),yCoor,zCoor));
        counter := counter + 1;
      );
    );
  );
  counter := 0;

  --Send non-active UAVs to a specific location
  for all component in set registry do (
      component.enforcePolicies(new Coor(0,0,0));
  );
);
```

Fig. 13. Self-* Mechanisms: Dependability policies to enable self-adaptive UAV swarm

5.3 Verification phase

We simulate the multi model and observe whether the MetaSelf functionalities meet the requirements stated in Section 5.1. Fig. 14 shows the actual x coordinate of all the UAVs in the swarm against simulation time. The graph shows that initially UAVs1-4 were coordinated around the x axis until around time 14 when UAV3 is replaced by UAV5. This shows that the first requirement has been met whereby backup UAVs are able to replace low battery UAVs. UAV2 eventually ran out of battery at around time 25, and the remaining UAVs spread out accordingly to maximize coverage area, suggesting that the UAVs can adapt to the number available, meeting the second requirement.

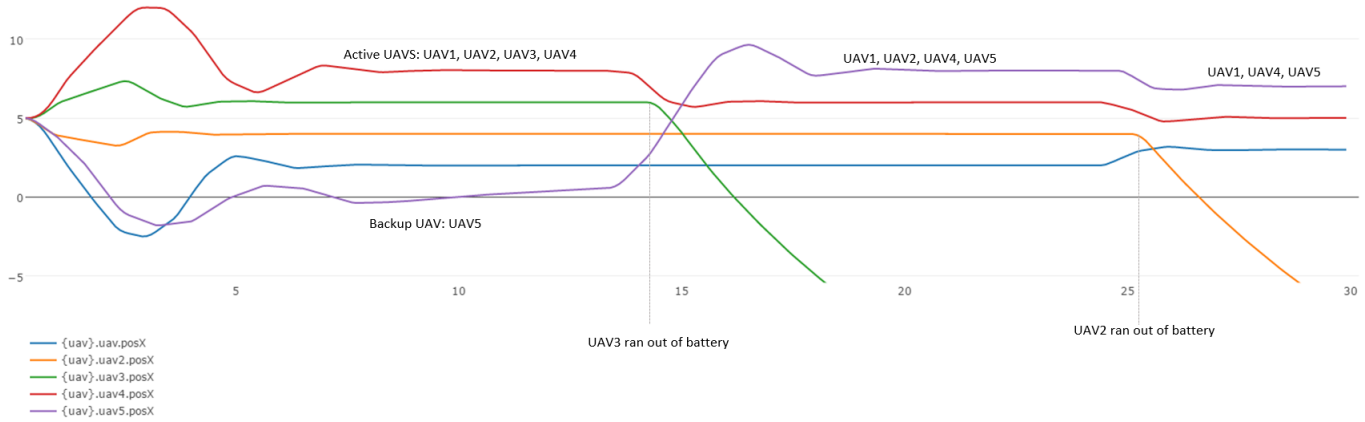

Fig. 14. INTO-CPS UAV swarm simulation graph showing the actual x coordinate of all the UAVs

6 Conclusions

Here we discuss the feasibility of our approach in addressing several challenges in the design of self-* CPSs as briefly mentioned in section 2: *Cyber and Physical integration*, *Heterogeneous components*, *Level of Autonomy*, and *Real time*.

In our study, the elements that enable self-* are discrete-event processes that receive metadata and perform physical reconfigurations; the challenge lies in capturing both discrete and continuous dynamics. Our approach addresses this by separating the design of discrete and continuous processes into separate models. This means that, provided the designer is able to create a multi-model that captures the properties of physical processes into metadata, the discrete-event policies can describe appropriate reconfiguration actions.

In our example, the swarm is homogeneous. However, we suggest that our approach can readily support heterogeneous structures. Both Metaself and INTO-CPS follow component-based approaches specifically intended to support the design of heterogeneous systems. Their integration means that on the system level, different components can communicate as long as the metadata are properly defined. On the architecture level, different components can be designed as separate FMI-conformant communicating FMUs.

Metaself allows some autonomy, but has not yet been defined to accommodate models of machine learning (ML). The Metaself process suggests that guiding policies defined at design time will not change after deployment, but an ML-enabled CPS would likely have dynamically mutable policies. This might be achieved my adding a service layer in the infrastructure that monitors performance and modifies the guiding policy accordingly.

In our proposed approach reconfiguration actions will have to go through cyber, network and physical processes. The inability to schedule real-time performance from policy decision to execution makes this form of Metaself unsuitable for real-time CPSs. Metaself should be enhanced with a form of real-time scheduling to ensure reconfiguration in real time performances.

The applicability of our approach for self-* CPSs is limited by its lack of support for real-time operation, and further work is required to demonstrate its application to CPSs with heterogeneous elements. However, it has shown potentia as it addresses certain challenges in self-* CPS design, and we anticipate that solution to some weaknesses can be added on the existing infrastructure. Further work is also required to assess its feasibility on a wider range of CPS characteristics such as human-in-the-loop and security that are not yet addressed.

Acknowledgements We acknowledge assistance from the EC's INTO-CPS project (H2020 Project 644047) and are grateful to the Overture Workshop reviewers for their comments on our earlier draft.

References

1. Antsaklis, P.: Goals and Challenges in Cyber-Physical Systems Research Editorial of the Editor in Chief. IEEE Transactions on Automatic Control 59(12), 3117–3119 (Dec 2014)
2. Berns, A., Ghosh, S.: Dissecting Self-* Properties. In: Third IEEE Intl. Conf. on Self-Adaptive and Self-Organizing Systems. pp. 10–19 (Sept 2009)
3. Bharad, P., Lee, E.K., Pompili, D.: Towards A Reconfigurable Cyber Physical System. In: 2014 IEEE 11th Intl. Conf. on Mobile Ad Hoc and Sensor Systems. pp. 531–532 (Oct 2014)
4. Dai, W., Huang, W., Vyatkin, V.: Enabling plug-and-play software components in industrial cyber-physical systems by adopting service-oriented architecture paradigm. In: IECON 2016 - 42nd Annual Conf. IEEE Industrial Electronics Society. pp. 5253–5258 (Oct 2016)
5. Di Marzo Serugendo, G., Fitzgerald, J., Romanovsky, A.: MetaSelf: An Architecture and a Development Method for Dependable Self-* Systems. In: Proc. 2010 ACM Symp. Applied Computing. pp. 457–461. SAC '10, ACM, New York, NY, USA (2010)
6. Duarte, R.P., Bouganis, C.S.: Variation-Aware Optimisation for Reconfigurable Cyber-Physical Systems, pp. 237–252. Springer International Publishing, Cham (2016)
7. Elgenedy, M.A., Massoud, A.M., Ahmed, S.: Smart grid self-healing: Functions, applications, and developments. In: 2015 First Workshop on Smart Grid and Renewable Energy (SGRE). pp. 1–6 (March 2015)
8. Feljan, A.V., Mohalik, S.K., Jayaraman, M.B., Badrinath, R.: SOA-PE: A service-oriented architecture for Planning and Execution in cyber-physical systems. In: 2015 Intl. Conf. on Smart Sensors and Systems (IC-SSS). pp. 1–6 (Dec 2015)
9. Gao, X., Ai, X.: The Application of Self-Healing Technology in Smart Grid. In: Power and Energy Engineering Conference (APPEEC), 2011 Asia-Pacific. pp. 1–4 (March 2011)
10. Jatzkowski, J., Kleinjohann, B.: Towards Self-reconfiguration of Real-time Communication within Cyber-physical Systems. Procedia Technology 15, 54 – 61 (2014), 2nd Intl. Conf. on System-Integrated Intelligence: Challenges for Product and Production Engineering
11. Leitão, P.: Towards Self-organized Service-Oriented Multi-agent Systems, pp. 41–56. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
12. Park, S.O., Do, T.H., Jeong, Y.S., Kim, S.J.: A dynamic control middleware for cyber physical systems on an IPv6-based global network. Intl. Jnl. of Communication Systems 26(6), 690–704 (2013)
13. Payne, R., Gamble, C., Pierce, K., Fitzgerald, J., Foster, S., Thule, C., Nilsson, R.: Examples Compendium 2. Tech. rep., INTO-CPS Deliverable, D3.5 (December 2016)
14. Ratasich, D., Höftberger, O., Isakovic, H., Shafique, M., Grosu, R.: A Self-Healing Framework for Building Resilient Cyber-Physical Systems. In: 2017 IEEE 20th intl. Symp. on Real-Time Distributed Computing (ISORC). pp. 133–140 (May 2017)
15. Ruiz, A., Juez, G., Schleiss, P., Weiss, G.: A safe generic adaptation mechanism for smart cars. In: 2015 IEEE 26th Intl. Symp. on Software Reliability Engineering (ISSRE). pp. 161–171 (Nov 2015)
16. Schuh, G., Pitsch, M., Rudolf, S., Karmann, W., Sommer, M.: Modular Sensor Platform for Service-oriented Cyber-Physical Systems in the European Tool Making Industry. Procedia CIRP 17, 374 – 379 (2014)
17. Seiger, R., Huber, S., Heisig, P., Assmann, U.: Enabling Self-adaptive Workflows for Cyber-physical Systems, pp. 3–17. Springer International Publishing, Cham (2016)
18. Thomson, H., Paulen, R., Reniers, M., Sonntag, C., Engell, S.: D2.4 Analysis of the State-of-the-Art and Future Challenges in Cyber-physical Systems of Systems . CPSoS Project Deliverable, CPSoS (2015)
19. Yim, M., Shen, W.M., Salemi, B., Rus, D., Moll, M., Lipson, H., Klavins, E., Chirikjian, G.S.: Modular Self-Reconfigurable Robot Systems [Grand Challenges of Robotics]. IEEE Robotics Automation Magazine 14(1), 43–52 (March 2007)