A Programming Language for Sound Self-Adaptive Systems

Barry Porter
School of Computing and Communications
Lancaster University, UK
Email: b.f.porter@lancaster.ac.uk

Roberto Rodrigues Filho
Institute of Informatics
Federal University of Goiás, Brazil
Email: robertovito@ufg.br

***Abstract*—The ability for systems to *adapt* at runtime by hot-swapping their logic, seamlessly and without any apparent interruption, allows a program to adjust its behavior to its context. Research in adaptive systems support has to date focused on the basic mechanics of hot-swapping code at runtime, with the *soundness* of a system after each hot-swap left to the developer to assure on a case-by-case basis. Providing this assurance in existing programming languages is sufficiently difficult that self-adaptive systems using hot-swapping remain largely untrusted for production use. In this context we study two research questions: (i) what is the general soundness principle for self-adaptive systems; and (ii) how can we embed this soundness principle in a general-purpose programming language? We answer these questions partly by theoretical analysis, and partly through developing a novel general-purpose programming language which embeds our soundness principle – allowing any module to be hot-swapped with the soundness of the wider system guaranteed.**

I. Introduction

One-size-fits-all algorithms and systems generally fail to meet the challenges of dynamic operating environments, causing performance shortfalls compared to ideal implementations in each environment. This is true at a micro-level for single algorithms, such as buffer and queue management policies, and at a macro level in the composition of entire systems.

Self-adaptive systems capture these scenarios using *runtime decision-making* to dynamically infer which behaviors best match each context experienced by a deployed system. These systems work by connecting runtime decision logic with the ability to hot-swap different alternative modules in and out of a system, where hot-swapping is implemented within an existing programming language (e.g., [1], [2], [3]). However, because existing languages are not designed to support *sound* hot-swapping, programmers are forced to manually reason about soundness on a case-by-case basis, for every hot-swappable module, in order to avoid catastrophic system failures [4], [5]. This is a surprisingly complex task which requires a deep understanding of the possible risks of hot-swapping in a given language. Manually providing this assurance in existing languages is sufficiently hard that self-adaptive systems using hot-swapping remain largely untrusted.

In this context we study two research questions:

1) What is the general soundness principle for self-adaptive systems that use hot-swapping?
2) How can we embed this soundness principle in a general-purpose object-like language?

Our methodology to answer the first of these questions is a theoretical analysis of the soundness problem in self-adaptive systems, informed by existing research in hot-swap soundness problems within languages like Java [4]. This analysis yields a key principle that we must uphold, which is the avoidance of *un-testable hybrid state machines*.

For the second question our methodology has been to design and implement a new programming language, and test the generality of that language across a wide range of systems. Over the last 8 years we have developed the general-purpose object-like language *Dana* which is designed to uphold our soundness principle[1]. We have tested the generality of Dana by building large-scale systems from data centre back-ends [6] to big data processing frameworks [7], databases, IoT frameworks, and GUI toolkits. We derive a proof sketch to demonstrate how Dana upholds our general soundness principle *by construction* – meaning the rules of the language allow this property to be guaranteed for any program, where any module can be hot-swapped with soundness guarantees for the overall system in which that module exists.

II. Related Work

The designs of programming languages are based on *soundness theorems*, which allow certain properties of programs written in those languages to be formally proven. The most common soundness theorem is *type soundness* [8], typically realised as syntatic type soundness [9], which informs the design of a wide range of languages like Java and C++.

Soundness for runtime hot-swap of logic is rarely a language design goal; popular languages thus offer little assurance of well-defined behaviour after hot-swaps. Despite this, our work builds on a long history of research on runtime code updates.

Kramer and Magee undertook seminal work on *quiescence* [10] for distributed systems with transactions, demonstrating how a node to be hot-swapped can be placed into a consistent (but inactive) state. This was later refined by Vandewoude *et al* with the concept of *tranquillity* for lower disruption [11].

Local systems have different assumptions to distributed ones, including zero disruption between hot-swaps, and higher levels of state machine entanglement between system elements. In local systems the concept of *dynamic interposition*, introduced in the operating system K-42 [12], is sufficient for

[1] http://www.projectdana.com

the safety of the *individual entity being swapped*. This blocks all calls that would transit into an entity to be hot-swapped and places those calls in a queue, transfers state (if any) between the outgoing and incoming entity, then resumes queued calls. We use dynamic interposition in a similar way to K-42, but also show this is not sufficient for wider system soundness.

Beyond hot-swap mechanics, component frameworks with reflective runtime models such as OpenCom [1] and Fractal [2] support self-reasoning for architectural adaptation, with a runtime basis for meta-level operations on that architecture. We use many of the component-based ideas of OpenCom, though interfaces in Dana can be instantiated as objects and object references passed around – supporting a form of internal structure that is not controlled by an external meta-level.

A range of language-level research has more recently demonstrated that, for local systems, dynamic interposition and component models are not sufficient for sound hot-swaps. While they provide protection for the entity being swapped, the wider system can experience state machine errors which lead to general exception faults and system exit. In particular, dynamic Java class update toolchains such as JVolve [5] and Javeleon [13] show the extent to which Java classes can be hot-swapped at runtime, and which kinds of changes may cause system failures at a JVM-level. Our research takes particular inspiration from that of Gregersen *et al.* in this area [4] on the implications of stateful side-effects.

III. Designing a Sound Adaptive Language

In this section we first present the key theoretical problem involved in hot-swapping code at runtime, and from this derive a soundness principle in answer to Research Question 1. We then define the programming model of Dana, and its reference-passing model which supports our soundness properties.

A. The Adaptation Game

We can summarise the key problem in hot-swaps as a lack of *state machine equivalence*, leading to a system arriving in a *hybrid* state machine *which could never have been tested*. Because such a state machine could never have been tested, we are unable to make any claims or assurances about the behaviour of the system from this point onward.

Our soundness derivation is inspired by the classic work by Hoare [14], in which properties of programs can be proven by deductive inferences over a set of language axioms (this is the basis of type soundness theorems). In addition to these axioms, program correctness is in practice established using deductive reasoning over source code, combined with execution-based testing, to provide sufficient evidence that a program in state s_n will transition to state s_q as a result of receiving input i. The evidence that these transitions occur under the expected conditions gives assurance that a program is correct to its specification. It is then assumed that since a deployed program is equivalent to that which was tested, it must also be correct.

Our analysis assumes the existence of *multiple different* compositions of the same program, each of which has been individually subjected to the same tests to verify correctness. This is a generalisation of the classic hot-swap scenario, in which one module m_i in the currently-executing system has a replacement alternative module m_j loaded into memory, and all links in the system that reference m_i are updated to instead refer to m_j, after which m_i is unloaded from memory. In our generalisation, we assume two different compositions CC_a and CC_b, representing the same overall program, but with one or more sub-modules in CC_b that use a different implementation compared to that of CC_a (such as a different sorting algorithm, or a different scheduling algorithm).

We also assume that a computer program, though its source code, represents a set of *reachable states*. The actual states reached depend on the state machine described by the program text, plus the specific inputs received during execution. We can also say that the program text alone describes an *abstract* set of states SA that are reachable given *any* set of inputs.

In an adaptive system with hot-swaps, *each* available composition of behaviours CC_i has its own SA_i. In general, the set of states SA_i that are reachable within CC_i may be significantly different to the set of states SA_j that are reachable within a different composition CC_j – even though both compositions implement the same high-level functionality.

When we transition between two compositions CC_i and CC_j at time t during execution, we then have a problem: what happens if the state that CC_i is currently in does not exist anywhere in the SA_j of CC_j? More specifically, we must consider what happens if the transition from CC_i to CC_j at time t results in a *hybrid* state machine that exists *neither* in SA_i nor in SA_j. In this case the hybrid state machine *could not have been tested* and we can have no assurance about the properties of the executing program after this point. Compounding this problem, if we adapt between CC_i and CC_j at time k instead of time t, we may arrive at a *different* hybrid state machine that which was induced by adapting at time t. We therefore have an infinite number of possible hybrid state machines about which we can make no guarantees.

Without constraints, this effect makes hot-swapping inherently brittle: in platforms like Java, for example, general exception faults may occur in the *language runtime* due to type system errors caused by hybrid, untestable state machines [4]; above the language runtime, fatal faults may occur in programs generally due to object reference graph incoherence [3]. These potential effects – which are difficult to reason about in all but the simplest of systems – force engineers to work with extreme caution, and engender a lack of trust in hot-swapping.

We propose that a programming language designed with hot-swap soundness guarantees will minimally assure that:

When adapting from a composition CC_i to CC_j at any point in time, the system in CC_j must be in a state s_j that actually exists in the abstract state set SA_j of CC_j – and is therefore within an envelope that could have been tested.

A more strict guarantee is that the post-adapted system in CC_j at time t is in *exactly* the same state as if the system had *always* been in CC_j since the start of its execution; our experience suggests this is much more costly to assure and may require the use of a less general programming language.

HashMap.dn

```
data Item {
   char key[]
   Data value
   }

interface HashMap {
  //per-instance transfer state
  transfer Item content[]

  HashMap()
  void put(char key[], store Data val)
  Data get(char key[])
  }
```

Button.dn

```
interface Button extends UIObject {
  //per-instance transfer state
  transfer int xPos
  transfer int yPos
  transfer char label[]

  eventsource click()

  Button(char label[])
  void setPos(int x, int y)
  }
```

DefaultHashMap.dn

```
data HashBucket {
  KeyVal v
  }

data KeyVal {
  char key[]
  Data value
  KeyVal nxt
  }

component provides HashMap(AdaptEvents) {
  HashBucket map[] //internal state
  int count = 0

  void HashMap:put(char key[], Data val) {
    KeyVal kv = new KeyVal(key, val)
    kv.nxt = map[hash(key)].v
    map[hash(key)].v = kv
    count ++
    }

  void AdaptEvents:inactive() {
    content = new Item[count]
    int j = 0
    for (int i = 0; i < map.length; i++)
      for each (KeyVal k in map[i].v) {
        content[j] = new Item(k.key, k.value)
        j ++
        }
    }
  }
```

BasicButton.dn

```
component provides Button
          requires Font {
  Font font //internal state

  Button:Button(char text[]) {
    label = text
    font = new Font(Font.default, 12)
    }

  void Button:setPos(int x, int y) {
    xPos = x
    yPos = y
    }

  void Button:paint(Canvas c) {
    c.drawRect(...)
    c.drawText(font, label, ...)
    }

  void Button:mouseUp(int x, int y) {
    emitevent click()
    }
  }
```

Fig. 1. Example interfaces and components, showing both kinds of instance state. Objects can read/write to both *transfer* and *internal* state, but both kinds of state are invisible to other objects.

We next introduce the key design points of the Dana language. Together, these allow us to deductively assure that any system and any hot-swap must inherently meet our above soundness objective – by virtue only of each part of that system being successfully compiled and tested in isolation. Following any hot-swap to a composition CC_j at runtime, the resulting system must therefore arrive in a state that exists in the SA_j of CC_j – and must be equivalent to a program that has been tested and is known to be correct. With one exception, which we discuss in Sec. IV-C, this consequently removes the need for the programmer to consider the system soundness risks of hot-swaps when developing their code.

B. The Dana language

Dana is a general-purpose, imperative, object-oriented language designed for full-stack development. It is implemented with a custom-built interpreter and compiler which work together to assure hot-swap soundness: the compiler performs static checks where possible, and the interpreter performs additional dynamic checks during execution to prevent certain design patterns that would violate soundness guarantees.

Dana uses strong typing with two distinct type hierarchies: *interface types* for behavior, and *data types* for data, both of which support single inheritance. An interface type is a tuple of a set of function prototypes, a set of event prototypes, and a set of transferable state fields. Transferable state fields are private to the implementation of the interface for both read and write, and are declared in the interface only to ensure transfer state compatibility between alternate implementations. Because interfaces only have function prototypes, such that no publicly accessible writable state is available in any object, we can cheaply intercept all state transitions in a system by tracking function calls into objects.

A data type is simply a collection of named, typed fields in which to store values. Dana also provides two primitive types, integers and decimals, of various sizes, and an array type $A_{<t>}$ which is parameterised by any of the above types. Interface types are instantiated as objects; data and array types are also instantiated, with instances of all three types being passed by reference. Primitives are passed by value.

An interface I is implemented by a component C, such that C provides an implementation of every function prototype declared in I. Each object instantiated from an interface I has a new copy of I's transfer state, and C can also declare internal per-instance state specific to the implementation. As well as *providing* implementations of interfaces, components can also declare *required* interfaces, which represent dependencies on external behavior. At runtime, the required interfaces of each component are wired to compatible provided interfaces on other components to satisfy dependencies.

Example components, with associated interfaces, are shown in Fig. 1, in which we highlight both transfer state on interfaces and internal state in implementations. The `HashMap` interface declares a transfer state form which is different to the internal state representation of the implementing component. For this reason the implementation also declares an `AdaptEvents` interface, through which it is notified of adaptations and can convert state into/out of the transfer state format and its own internal representation. The user-interface `Button` example, by comparison, has only primitive transfer state and so needs no special functions to be inherently hot-swappable.

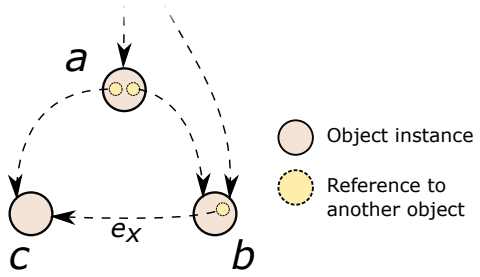

Fig. 2. An object reference graph which may yield an incoherent hybrid state when object a's logic is hot-swapped.

Altogether, a Dana system is formed by loading suitable components and wiring each required interface to a type-compatible provided interface (wirings are in practice represented by lists of function pointers). Objects are instantiated via required interfaces, where each object o comprises $\{i, t_s, c, c_s\}$: a provided interface type i (that via which it was instantiated), a copy of the transfer state fields t_s of i, the linked logic of a component c which currently implements that object, plus an instance of the private state c_s declared in c for that object. An object's implementation can be changed during a runtime hot-swap to a different component c', resulting in o becoming $\{i, t_s, c', c'_s\}$. All references to the object o remain valid after a hot-swap of its implementation, with only o's internal function pointers and state changing.

C. Reference-passing rules

The use of strict encapsulation, and explicit dependencies via required interfaces, are necessary but not sufficient to support sound hot-swapping by construction. To fully support this, Dana also enforces restrictions on how object references can be passed, and uses an event model to cover design patterns that would otherwise be impossible under these restrictions.

Unrestricted object reference graph formation can make hot-swap soundness impossible as follows. Consider three objects a, b, and c, each sourced from three different components, as shown in Fig. 2. Object a is given a reference to b; a then instantiates c via one of its required interfaces, and passes a reference of c into b. The implementation of a, which added the edge e_x to the reference graph, is then hot-swapped to a different implementation a' which does not include the logic to pass the reference of c into b. Because the logic which did this has left the system, and b is still present in the system, we have a hybrid state machine composition which could not have been reached had the system always been in its new composition from start – and we are unable to make any assertions about the subsequent behavior of the system as its hybrid state machine has not been tested. This may result in behaviours ranging from 'strange' [4] through to nonsensical runtime exceptions.

Dana therefore restricts the object reference graphs that can be formed. First, we note that object references can be passed freely without restriction into any object o_p if those references are not going to be *stored* in o_p's internal state (i.e., are only used within the local scope of the called function). We can do this because references that are only used within the scope and lifetime of a called function do not persistently affect the reference graph. If a reference *is* going to be stored in the object's state then Dana enforces a rule that an object a can only pass such object references into objects that a itself instantiated. This supports the most common kinds of object composition (such as instantiating a window and then adding a set of buttons to the window), while avoiding scenarios in which component n can be hot-swapped out of a system after having created edges in the object reference graph that persist following its departure. Programmers declare at the interface level whether or not an object will store any function parameters of type `Data` or `Object`, as exemplified on the `put` function of the `HashMap` interface in Fig. 1. In the majority of cases the Dana compiler can automatically detect when the programmer needs to declare parameters as storable, introducing little extra programming complexity.

Finally, to support design patterns which would otherwise be unavailable under the above restriction (particularly those in which objects would register for notifications from other objects), Dana provides an asynchronous event paradigm whereby an interface can declare events that it emits, such that an object o_n can listen to events from another object o_j. An example event provider is the `Button` component shown in Fig. 1 which emits `click` events to interested listeners. Objects cannot see *who* is listening to their events, and can only emit an event to 'all listeners', allowing us to freely adjust registrations to uphold soundness properties (see Sec. IV).

D. Hot-swap protocol

When performing a runtime hot-swap, the required interface of a selected component is re-wired during execution to point to the provided interface of an alternative implementation. We first 'pause' the required interface, causing any future object instantiation instructions to be blocked and placed into a queue. We then iterate over each extant object instantiated over this required interface, and 'pause' that object such that future function calls are blocked and placed into a queue. For stateful objects, we also wait for any in-progress function calls to finish. We then allow the outgoing implementation of an object to translate its internal state representation to the transfer state definition on the interface, and the incoming implementation of that object to translate from this common state representation to its internal representation. The object is then un-paused, such that any queued function calls may continue into the object; when all objects have been adapted, the required interface itself is re-wired and un-paused so that any queued instantiation instructions may proceed.

IV. Soundness derivation

Dana provides inherent system soundness properties after any code hot-swap. Formally proving theorems for a full-featured, general-purpose language would exceed the space available here; in this section we therefore provide a sketch of how our soundness properties are derived.

Our sketch assumes the existence of multiple different compositions of the same program, each of which has been individually subjected to the same tests to verify correctness. We then show that, when adapting from one composition of components CC_i (with potential state machine SA_i) to another CC_j (with potential state machine SA_j), the adapted-into system is *not* a hybrid state machine but rather must be in a state $s_j \in SA_j$. Our demonstration that this is always true, for any hot-swap, relies on understanding the way in which Dana constrains state machine transitions in general.

We first show this property in a simpler model than that of Dana, then introduce each complexity the full Dana language adds to this model and extend the demonstration accordingly.

A. Simple model

Our simple model assumes there are *no inter-object side-effects* in any state transitions: a state transition that occurs in any object q induces no state transitions in any other objects.

In this model we focus on a single hot-swap of a component x to x', to move from composition CC_i to CC_j. Because we are only changing a single component, every *other* component in CC_j is evidently the same as in CC_i. As there are no side-effects between objects, it follows that the state machine SA_j must be identical to SA_i in every component except x'. It also follows that the specific state of *the rest of the system* must exist within SA_j, and so the rest of the system must be correct following the hot-swap with well-defined behaviour.

We can then focus on the state machines of components x and x'. If x and x' are stateless we can deduce the entire system must be correct after the hot-swap, relying only the per-component protocol described in Sec. III-D.

If x and x' *do* have state we rely on the state transfer features of Dana in which each interface can declare one or more *transfer* state fields, representing elements of object state which must be transferred when components are hot-swapped. Using this mechanism we then require the following assertion to uphold soundness: the specification and correct implementation of an interface implies that any component x is able to (i) convert its internal state to the transfer state in its interface, and (ii) convert from the transfer state to a *valid state* in its internal state machine. With this assertion, we can then say that x' must have arrived in a valid state and so the entirety of CC_j as defined above must be in a state $s_j \in SA_j$, and so is correct following a hot-swap.

Using our `AdaptEvents` interface to notify components they are being hot-swapped out of / into a system, transfer state thus acts as a conduit to reach valid states in the internal state machines of different implementations of the same interface.

B. Full Dana model

We next show that we can also uphold our objective in our full language model, and so similarly assure that a system with hot-swaps is correct. To fully model the state machine space of Dana we enhance our model to include the potential for *side-effects*, where the state transitions in one object can cause state transitions in other parts of a system. This can occur in three ways: (i) the object reference graph; (ii) the registrations on event sources; or (iii) the internal state of objects. In the remainder of this section we cover each of these in turn.

1) Reference graph: The object reference graph is a globally shared structure between all objects, since any object can modify the nodes and directed edges in that graph. In Dana, an object is only permitted to add edges to the reference graph which point *from* objects that it has created (*to* any object).

From this rule we can deductively see that when an object leaves the system, all of the objects (nodes in the reference graph) it created must also leave the system. Likewise, all edges in the reference graph that an object created will also be removed when it leaves the system.

In other words, an object cannot affect the reference graph outside of its own created sub-graph. In this sense, the reference graph upholds the property that no side-effects are allowed, because when component c leaves the system the reference graph is returned to a state that would have existed had c never been present. Our above derivation for a model without side-effects therefore holds for this added complexity.

2) Event registrations: Objects in Dana can register to receive events from any other object. The *list of registrations* on any given object is therefore a vector for side-effects. However, when an object leaves the system (including when it is being hot-swapped out), all of its registrations can be safely deleted because objects have no way to access the list of registrations on their event sources. This returns the system's event registration state to a point where the object in question was never present, and so again upholds the above derivation for our simpler model without side-effects.

3) Programmer-defined state: Finally, an object x can call functions on other objects, where those calls modify the state of those objects. x may then be hot-swapped to x' which would not have made the same sequence of function calls.

Our avoidance of untested hybrid state machines here relies on a property we term *permissive side effects*, which all object-oriented languages exhibit, and which forces the programmer to accept arbitrary state transitions under certain conditions.

We first observe that, in an object-oriented language, when one object x creates another y, and passes a third object z into y, x gives y *permission* to drive arbitrary state changes in z.

As a real example, consider the simple program in Fig. 3, in which we instantiate a `JSONParser` object, then a `File` object, and pass the file object into the parser for reading. Here the source code represents object x; the JSON parser is object y and the file object is object z. Here the programmer of object x must program in such a way that x can accept arbitrary state transitions in z. This is reinforced in Dana because every dependency is an abstract interface, so the programmer has no details about the implementation of an interface (aside from any semantics implied by the interface definition).

Now consider the hot-swap of each of the objects in our above example in Fig. 3. The hot-swap of object x to x' causes the destruction of objects y and z (as there are no other references to these objects) so we can ignore their state; for x itself we assume that the state transfer process will derive a

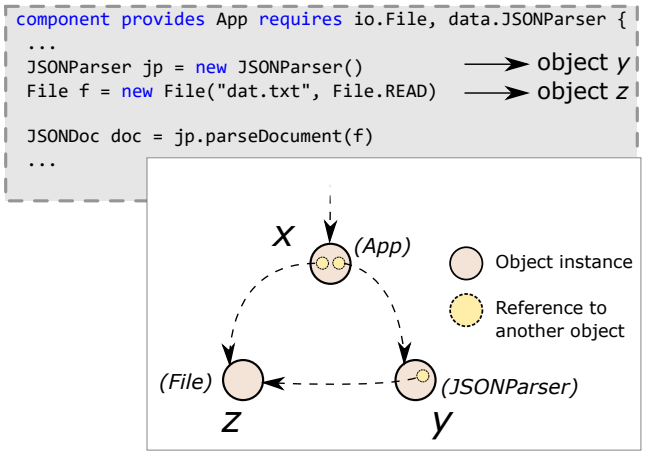

Fig. 3. Permissive stateful side effects when passing object z into y.

state for x' which is within its set of valid states and so will be correct (see Sec. IV-A). The hot-swap of object z has the same properties: state transfer will derive a valid state for z', and z itself did not have access to any other object references and so could not have affected the rest of the system.

The hot-swap of object y partly operates under the same state transfer case as x and z: the only object with a reference to y is x, and thus x has been driving all state transitions in y. As long as y' has a state that is internally valid to the logic of y', and is coherent with x's understanding of what its state should be, x and y' will exhibit mutually compatible and coherent states. However, y has also been able to arbitrarily modify the state of z, and when y is hot-swapped out it may have made changes to the state of z which y' would not have.

In this scenario we can consider the validity of the state of z with respect to the two other objects which may interact with it: y' and x. First, for the state of z relative to y', our state transfer assertion requires the transfer from y to y' to yield an internally valid state for y'; if the interface of y/y' implies that they store a reference to z, our state transfer assertion must by induction include the state of z for y'. In other words, the implementation of y' is necessarily forced to either accept any state of z, or to move z into a state acceptable to y'. Second, the state of z relative to x must be valid since x has given *permission* for the state of z to be modified in an arbitrary way; since x must be programmed without awareness of the specific implementation of y, x must necessarily be programmed to operate correctly for *any* state of z.

In each case, the hot-swap of any of these three objects in a transition to composition CC_j must cause the resultant system state s_j to be within SA_j, and thus the system must be correct after hot-swaps despite these stateful side-effects.

C. Exposing Internal Implementation-Specific State

There is one edge case for soundness which Dana does not explicitly cover, for return values / event data. An object x can return a data instance or array instance that x created[2], where the content of that instance is specific to the implementation – such that hot-swapping x to an alternative implementation of the same object x' would leave a piece of state in the rest of the system that could not have existed if x' had been present since system start. While our current implementation does not cover this, the most obvious solution is to annotate such a return value to indicate that it is implementation-specific, and when a hot-swap is requested wait until there are no references to those entities present in the rest of the system outside of their creating object (or objects that it created). Our experience to date suggests this scenario is rare, since abstract interfaces (and what they return) tend to be inherently defined in a non-implementation-specific way; we leave further examination of this edge case as a topic of future work.

[2]Note that such data or array instances cannot, directly or indirectly, reference any objects, as implied by Dana's reference-passing rules.

ACKNOWLEDGMENTS

This work was partly supported by the UK Leverhulme Trust via the Self-Aware Datacentre project, grant RPG-2017-166.

REFERENCES

[1] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan, "A generic component model for building systems software," *ACM Trans. Comput. Syst.*, vol. 26, no. 1, pp. 1:1–1:42, 2008.

[2] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani, "An open component model and its support in java," in *Component-Based Software Engineering*, ser. LNCS, vol. 3054. Springer Berlin Heidelberg, 2004, pp. 7–22.

[3] F. Shen, S. Du, and L. Huang, "A dynamic update framework for OSGi applications," in *High Performance Computing and Applications*, W. Zhang, Z. Chen, C. C. Douglas, and W. Tong, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 350–355.

[4] A. R. Gregersen and B. N. Jørgensen, "Run-time phenomena in dynamic software updating: Causes and effects," in *Proceedings of the 12th International Workshop on Principles of Software Evolution*. ACM, 2011, pp. 6–15.

[5] S. Subramanian, M. Hicks, and K. S. McKinley, "Dynamic software updates: A vm-centric approach," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2009, pp. 1–12.

[6] B. Porter, M. Grieves, R. Rodrigues Filho, and D. Leslie, "RE^X: A development platform and online learning approach for runtime emergent software systems," in *Symposium on Operating Systems Design and Implementation*. USENIX, November 2016, pp. 333–348.

[7] P. Dean and B. Porter, "The design space of emergent scheduling for distributed execution frameworks," in *Proceedings of the 16th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE, 2021.

[8] R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and System Sciences*, vol. 17, pp. 348–375, 1978.

[9] A. Wright and M. Felleisen, "A syntactic approach to type soundness," vol. 115, no. 1, p. 38–94, Nov. 1994.

[10] J. Kramer and J. Magee, "The evolving philosophers problem: dynamic change management," *IEEE Transactions on Software Engineering*, vol. 16, no. 11, pp. 1293–1306, 1990.

[11] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt, "Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 856–868, 2007.

[12] C. Soules, J. Appavoo, K. Hui, R. Wisniewski, D. da Silva, G. Ganger, O. Krieger, M. Simon, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis, "System support for online reconfiguration," in *Proceedings of the USENIX Annual Technical Conference*, 2003, p. 141–154.

[13] A. R. Gregersen, B. N. Jørgensen, Hadaytullah, and K. Koskimies, "Javeleon: An integrated platform for dynamic software updating and its application in self-*systems," in *2012 Spring Congress on Engineering and Technology*, May 2012, pp. 1–9.

[14] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–583, 1969.