A Theory of Closure Operators

Alva L. Couch and Marc Chiarini

Tufts University, Medford, Massachusetts, USA
alva.couch@cs.tufts.edu, marc.chiarini@tufts.edu

Abstract. We explore how fixed-point operators can be designed to interact and be composed to form autonomic control mechanisms. We depart from the idea that an operator is idempotent only for the states that it assures, and define a more general concept in which acceptable states are a superset of assurable states. This modified definition permits operators to make arbitrary choices that are later changed by other operators, easing their composition and allowing them to maintain aspects of a configuration. The result is that operators can be used to implement closures, which can in turn be used to build self-managing systems.

1 Introduction

Cfengine[1–5] is a widely used tool for managing computing systems. Cfengine's basic building block is the "convergent operator", an operation that enforces a policy by modifying any non-conforming system state. Operators affect a broad range of system entities, including configuration and dynamic runtime state. Convergent operators *immunize* a system against potential deterioration, by repeatedly repairing any state found to be non-conforming. They are also *idempotent* on properly conforming systems, in the sense that they will do nothing unless nonconformity is discovered. Thus, each operator comprises a tiny autonomic "control-loop" that checks for policy conformity and implements changes if they are needed.

Primitive operators in the current Cfengine-II (and the upcoming Cfengine-III[6]) are simple in structure. It is best to think of Cfengine as an "assembly language" for building convergent operators. Statements in Cfengine's policies affect files, processes, and other entities in straightforward ways, and can be composed to accomplish high-level tasks such as "implementing services". These statements are implemented by local agents running on each managed system.

This paper is about the "next level". While Cfengine embodies Burgess' theories, it does not implement Burgess' more general theoretical definition of convergent operators[7, 8]. In that definition, a convergent operator assures that a system attribute has a value in some *set of acceptable values*, while most Cfengine operators assure that a system attribute has *one* value.

The effect of this limitation is that it is difficult for Cfengine to collaborate with other entities in assuring a goal[1]. Cfengine policies are centered around

[1] As Q of *Star Trek: the Next Generation* would say, "I don't work well in groups. It's difficult to work in groups when you're omnipotent."

creating some *specific* state, not an *acceptable* one. Thus, if some other entity creates another *acceptable* state, Cfengine will often detect and revert that state to the one and only state that it considers acceptable.

For example, suppose we set up a web server with Cfengine and then decide to tune its performance (either manually, or via some other software mechanism). Cfengine will – when invoked – revoke the changes we make to tune the server according to its own definition of "health," which is defined in terms of the contents and positions of specific files. To make the tuning "permanent", we have to inform Cfengine itself about the changes we want, and let *it* enforce these rules. This is an extra and potentially costly step if the changes are widespread.

In this paper, we ask the question, "how can convergent operators collaborate?" We explore collaboration as a composition of operators that does not require the step of coding knowledge from one operator (e.g., tuning) into another (e.g., setup). We define a new concept of operator that conforms to Burgess' theoretical definition but is broader than Cfengine's definition, and we explore the effects of composing such operators. We seek a situation in which two operators – one which sets up a web server and another that tunes it – can be composed and efficiently collaborate *without* knowledge of one another. This includes the human kind of operator as well. We explore a concept of convergent operator that encodes *intelligence* without requiring *rigid conformity* to a policy, so that the configuration agent becomes a "partner" rather than a controller. The desirable end result of this work is an autonomic control model that involves a partnership between humans and agents rather than a master-slave relationship in either direction.

How does one construct a well-designed convergent operator for network management? So far, the Cfengine model provides the only answer to this question. In this paper, we look beyond Cfengine's capabilities to a more general definition of convergent operators inspired by the theory of closures. All Cfengine operators comply with this definition, but the definition allows new kinds of operators to be created with desirable properties. Our conclusion is that this broader definition provides kinds of behavior that are otherwise difficult to describe or codify. In particular, while one can translate a policy to an operator, the converse seems intractable for some of the operators in this new class.

In the pages that follow, please keep in mind that fixed-point operators do not abandon the autonomic computing mechanism of closed-loop control; instead, they *encapsulate* control loops into smaller packages called operators. An operator includes a precondition-checking step that decides what control to apply, followed by an implementation step that makes appropriate changes. This can be viewed as a control loop operating *inside* the operator.

For example, consider two approaches to performance tuning of a web server, one based upon fixed-point operators and one based upon traditional control loops. The fixed-point version still contains a control loop, *inside* the operator, which is implemented through multiple invocations of the operator. This includes data collection, planning, and execution phases, but in the context of a single operator, rather than in the context of managing a whole system.

2 Convergent and Closure Operators

All the ideas in this paper presume the existence of a set of convergent operators $\mathcal{O}$ operating on a system that possesses a set of potential states S.

Definition 1. *A* convergent operator O *over a set of potential states S (that can be present in a network) is a function from S to S that, when applied repeatedly, eventually assures that a subset S^{a} of* assurable states *of S is present in the network, where O is idempotent over S^{a}, i.e., for $s \in S^{\mathrm{a}}, O(s) = s$.*

In other words, there exists some $k > 0$ such that for $n > k$ and any state $s \in S$, $O^n(s) \in S^{\mathrm{a}}$, or, equivalently, $O^n(S) = S^{\mathrm{a}}$ (and $O(S^{\mathrm{a}}) = S^{\mathrm{a}}$).

This is fairly close to the Cfengine definition of a convergent operator, but is more limited than Burgess' general definition of convergence, in which k might be infinite. Also, note that the existence of k in a static environment does not assume that there is a k for an environment that is dynamically changing, perhaps in opposition to the goals of the operator. For example, an operator that seeks to limit the number of user processes would never converge if a user attempted consciously to circumvent the operator by creating a steady supply of processes.

We broaden this definition in one fundamental way, inspired by the theory of closures, to admit a new kind of idempotence. A closure[9–11] is a domain of semantic predictability in a larger system that may exhibit unpredictability in other ways. Creating a closure requires separating behavior from configuration, so that configuration data can be classed as either *crucial* to behavior or *incidental*. An incidental configuration parameter's value does not affect behavior, while a crucial parameter changes observable behavior. This classification of parameters determines which configuration parameters should be part of the interface to the closure, and which should be internal and unexposed.

In particular, the first definition does not account for the fact that an operator may *accept* (that is, be idempotent over) more states than it *assures*. The *assurable* states of an operator are those that it has the power to create, while the *acceptable* states of an operator are those that it finds acceptable, but might not be able to create itself. More formally,

Definition 2. *A* closure operator O *over a set S is defined by two sets of states, $S^{\mathrm{a}} \subseteq S^{\mathrm{i}} \subseteq S$, so that O is idempotent over S^{i} but assures a perhaps smaller subset S^{a}.*

For most Cfengine operators, $S^{\mathrm{i}} = S^{\mathrm{a}}$, though there are some advanced operators (e.g., file editing) for which S^{i} is a proper superset of S^{a}.

The fact that assurance is different than acceptance is a core idea in the theory of closures. Many states are "acceptable" simply because they embody *arbitrary* (or "incidental") choices that do not affect the outcome of the goal. Other choices might be *crucial*.

For example, the actual location of the web server document tree has little to do with the basic behavior of a web server, so an operator O_1 that sets up a web server might well make an *arbitrary* choice[10] about those details. But one operator's "arbitrary" might be another operator's "crucial"; consider an

operator O_2 whose goal is to tune the performance of the web server. Then the choice of document root – unimportant to the basic act of setting up the web server – changes from incidental to crucial and is no longer so flexible. However, the first operator O_1 *does not care* about the location, so it should not override the operator O_2's changes to its original design.

The difference between what an operator accepts and what it assures can lead to fixed points that are not readily apparent. If O_1 *assures* that the web service is located on the mounted filesystem, but *accepts* that it could be located anywhere, then the set of operators $\{O_1, O_2\}$ has a fixed point. If O_1 does not accept anything except what it assures, then operators in the set share no fixed point and are not consistent. These two situations are described in Fig. 1. In one case, O_1 accepts states set up by O_2; in the other, it does not.

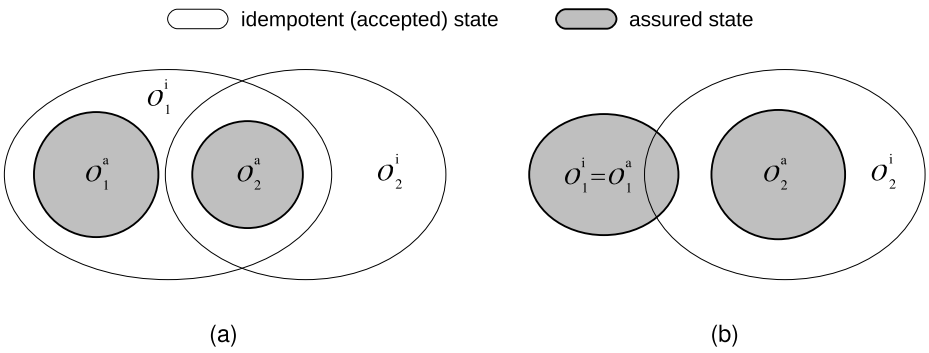

Fig. 1. (a) O_1 accepts some subset of states that are assurable by O_2. (b) O_1 only accepts the states that it assures.

Thus, the concept of incidental complexity in closures – conceived within a theory based upon syntactic consistency – is reflected in the theory of operators as an acceptance set that allows greater flexibility for incidental choices than those in an assurance set. A third operator O_3 might set up and assure the operation of a particular virtual server, without conflicting with O_1 or O_2.

3 Strategy and Tactics

One way of understanding this new distinction is that the acceptance set reflects a general *strategy* for accomplishing an aim or goal, while the assurance set reflects specific *tactics* for achieving that goal. A strategy is *declarative knowledge*, in the sense that it is about describing acceptable states, while a tactic represents *procedural knowledge* about how to *achieve* one or more assurable states.

In the above example, the strategy of O_1 is to "create a working web server," while the strategy of O_2 is to "create a fast web server." O_1 has a tactic for creating *at least one version* of a "working web server" in which it makes a

number of choices that might be incidental to the web server's function. To O_2, though, some choices (e.g., the locations of particular content) are no longer incidental, because O_2's strategy differs from O_1's strategy. However, a "fast web server" is indeed some subclass of a "working web server", so there is *strategy overlap* between O_1 and O_2. O_1 and O_2 can "collaborate" if O_2's tactic for creating a "fast web server" is *also* consonant with O_1's definition of a "working web server." In this case, we can *compose* O_1 and O_2 to get "a working and fast web server".

The concept of closure operators may seem abstract and unrealistic at first glance, but it is a more direct mimicry of what humans do in administering systems than the actions performed by current configuration tools. We break down installation of a service into multiple steps, each of which requires prerequisites. We tune each step separately, making sure we do not break the function of any prior step. One person might set up a web server, another might populate its content, and a third might tune it. This is exactly what a set of closure operators are meant to do, and one might thus characterize closure operators as "what humans do" to create and then continue to assure a functioning system.

The Cfengine way to construct O_1 and O_2 is to employ assurance sets that equal their acceptance sets. In that case, O_1 must embody all of the complexity of O_2, or the operators are inconsistent. If, instead, we can determine a method that allows O_2 to *embrace* what O_1 has done, without understanding it, and for O_1 to embrace what O_2 will do, then we can compose O_1 and O_2 into a whole greater than the sum of its parts.

4 Operator Consistency

One important question for a set of convergent operators (each of which has a set of fixed points) is whether the *set* shares a fixed point or not.

Definition 3. *A set of convergent operators $\mathcal{O}$ is* consistent *if the set shares a set of fixed points F that is a subset of all states S in the domain of the set of operators.*

Note that this is a set intersection problem: the set of fixed points F for a set of operators $\mathcal{O}$, if it exists, is the intersection of the sets of fixed points $\mathcal{A} = \{A_i\}$ that each individual operator O_i assures. $\mathcal{O}$ is consistent exactly when $\mathcal{A}$ is non-empty[2].

The above definition applies to operators with preconditions in a perhaps unexpected way. Such an operator does nothing until its preconditions are met. For example, one cannot tune a web server (O_2) until it has been installed (O_1). Thus O_2 is idempotent both when its preconditions are not met and after its acceptable states have been achieved.

Definition 4. *Operators with preconditions are consistent only if they exhibit a fixed point* after *all operator preconditions have been met.*

[2] Extended notions of consistency are explored in [12]

In other words, the trivial fixed point for operators that have not become active does not count as a fixed point for the set of operators.

It is often important to know whether consistency is a concrete or abstract property of a set of operators. It is only concrete if it can actually *arise* in practice:

Definition 5. *A set of operators $\mathcal{O}$ is* reachably consistent *(with respect to a set of baseline states B) if they are consistent, and for any state $b \in B$ of the network before the operators, there is some sequence of operator applications that leads to a consistent state.*

We denote the reachable states for a set of operators $\mathcal{O}$ with respect to a set of baseline (initial) states B as $\mathcal{O}^*(B)$.

Reachable consistency of closure operators is *not* a simple set intersection problem unless the acceptance and assurance sets for operators are equal; there are sets of closure operators that are consistent but not reachably consistent, because reachability requires some outside force to be applied. Consider the case where O_1 and O_2 share an *acceptable* point that is not *reachable*. For example suppose that O_1 can locate the web server in */usr/web*, O_2 can locate the web server anywhere in */opt*, and some third operator O_3 can put the web server in */var/www*, which is acceptable to both of the others. The set $\{O_1, O_2\}$ is *consistent* but not *reachably consistent*, because the common acceptable state */var/www* cannot be *reached* by virtue of the knowledge contained in O_1 or O_2. The set $\{O_1, O_2, O_3\}$ is *reachably consistent* (with respect to the set of all states S) because an assurable state of O_3 can satisfy at least one acceptable state of both O_1 and O_2. Thus a set that is not reachably consistent can be made reachably consistent by *adding* operators, a fact that is on the surface quite counter-intuitive and that cannot happen if assurable and acceptable states happen to match (Figure 2).

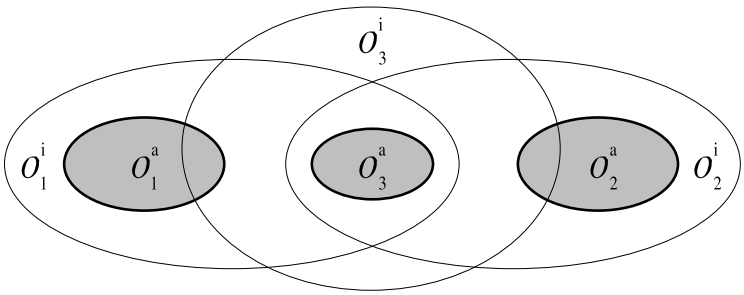

Fig. 2. Operators O_1 and O_2 can never reach a fixed point without O_3, which joins their acceptable states with a common assurable state. An 'i' (idempotent) superscript denotes an operator's set of acceptable states, while an 'a' denotes its assurance set.

Reachable consistency depends upon the possible initial states B of a system before operators are applied. This is a rather trivial assertion in the sense that

if the baseline states are fixed points of all operators, then the set of operators is consistent even though they might not be able to assure that particular state themselves. If, for example, the set of operators never moves the system outside a baseline state, then for all practical purposes, the set of all states *is* the baseline set.

In the above example, the result of the three operators is an *emergent* fixed point that is fixed for all three operators, but fixed in *different ways* for each operator. Thus each operator can be thought of as acting on an *aspect* of the network, and the composition of operators into a set can be viewed as similar to *aspect composition* as defined in [13–15]. An aspect is one facet of coordination within a configuration; any configuration management tool must *compose aspects* to create a valid configuration.

5 Implementing Closure Operators

One reason that closure operators have not been explored so far is that they are more difficult to implement than simple convergent operators. But there are things that simply cannot be done without them, including management operations that exploit aspect composition. Further, the extra machinery required to implement a closure operator is necessary *anyway*, for other reasons.

Closure operators are more difficult to construct than, e.g., automation scripts that accomplish similar goals. The closure operator, unlike a simple script, must be aware of its surroundings and have knowledge of its preconditions and post-conditions. An ideal closure operator is safe under all conditions, in the sense that it is safe to invoke the operator with the network in any conceivable state, and as a result, the operator will not damage the network through lack of knowledge (though it might not be capable of improving the network either).

Note first that the acceptance set of a closure operator can only be determined by use of a *validation model* that determines what is acceptable. This model is different from the *assurance model* that determines which settings will be changed if the current configuration is not valid. Both of these models could be specified as rulesets.

Consider, for example, how one would implement the operators O_1 and O_2 above. O_1 is straightforward enough; installing the RPM for Apache might do nicely as a first approximation. But O_2 is a much more sophisticated operator than has ever been written before. O_2 must *validate* the install of O_1 and then operate in such a way that this validation is not lost by its changes. This is a matter of *coordinating* settings in files with positions on disk, so that everything one moves is matched with a parameter change in a file. Further, for O_1 to accept this change, it must share with O_2 an underlying validation model that accepts more states than what it can assure. Thus the key to implementing O_1 and O_2

is that *both must agree (at some level) on what constitutes a valid web server, invariant of how that validity was reached*[3].

With this validation model in hand (ostensibly, modeled as a set of "valid web server configurations"), O_1 checks whether this model is satisfied, and takes steps to satisfy it if not. O_2, by contrast, does *nothing* if the web server configuration is invalid, but if it is valid, changes it to perhaps *another* valid state that responds more quickly.

Why would we want to structure operators in this way? One answer is that the monolithic construction of an operator that both installs and tunes a web server is more complex than two operators, each of which handles one aspect, and that there may in fact be different concepts and models of tuning that one might wish to apply. Further, the added complexity of a validation model is desirable whether we implement the tuning as one operator or two, because the alternative is that the web server may be unintentionally rebuilt for no particularly good reason, simply because "incidental" (and meaningless) changes in configuration have occurred out of band. In order to satisfy the spirit of Cfengine, that "if it isn't broken, don't fix it", one must have a model of what it means to be functional or broken.

6 Validation Models

What is a "valid" web server? This is a complex question that has been studied in some detail. First, *inside* the server, there are a set of data relationships that must be preserved. But there is another validation model that depicts how certain data must be present *outside* the web server. These are related via a closure model of web service[10] that expresses external behavior as a set of exterior maps. But in constructing this mapping, many *incidental* choices are made that have nothing to do with the mapping, though they may affect *performance*. These incidental choices must be *coordinated* so that the result is a functional web server. Thus there are two kinds of validation models: an *interior* model that depicts *data* (static) relationships, and an *exterior* model that depicts *behavioral* (dynamic) relationships.

An example of an interior model (reprinted from [10]) is given in Fig. 3. A web server configuration contains many parameters that must agree in order for the webserver to function properly. The directory in which content appears must both be accessible as a directory and mapped to an appropriate virtual server. Likewise, the name of each file must correspond to the appropriate MIME type, etc. Many of these parameters are "incidental" in the sense that choices for internal location of a set of files seldom affects the externally observable behavior of a web server.

The good news is that the internal model of a valid web server is a purely *declarative* description of data relationships, similar to database integrity con-

[3] Of course, RPM sets a flag that keeps a web server from being installed on top of another, but this also allows that web server to be manually broken and not repaired. So RPM does not really implement a closure operator.

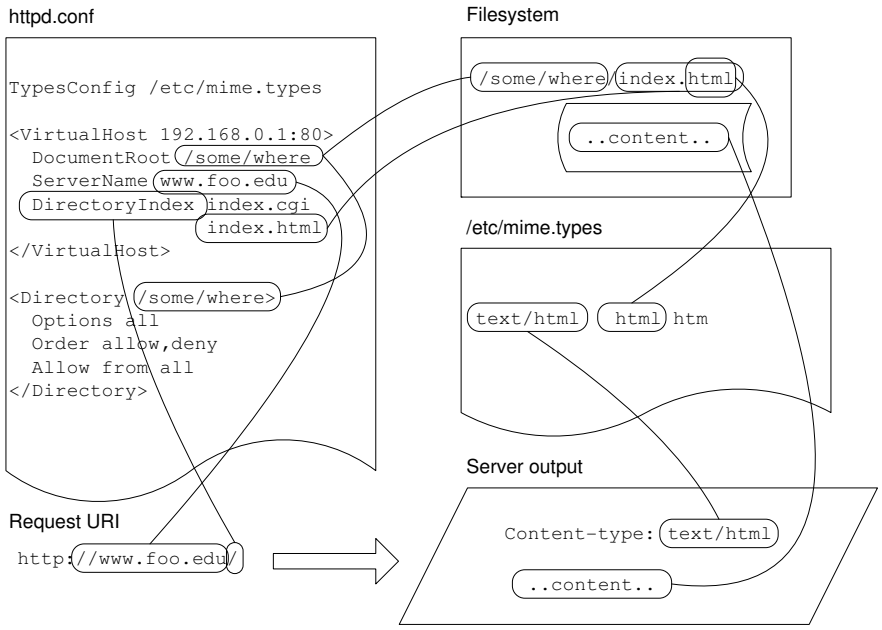

Fig. 3. A relational model of a specific web server's data dependencies (reprinted from [10]).

straints, and that it is both package-specific and policy-neutral, in the sense that any policy can be enforced while obeying the model. The bad news is that no such model has yet been constructed for many configurable packages in current use. This is a bad thing, because such models are necessary in order to know whether configuration management operators are working properly or not. An instance that does not conform to the basic model for behavior cannot possibly function properly.

Note that a model of validity is not an *information model* (e.g. CIM), but rather a *relational model*, similar to a set of database constraints or an XML schema. An overall model of a valid web server is (at some level) a version of the server's *user manual,* while a model of a "fast web server" is an embodiment of *best practices* for increasing service speed *while obeying the user manual.*

It might be best to think of each of these models as an XML document conforming to a particular XSchema; this seems to describe the models in the web server example, among others. The O_1 XSchema simply defines what is valid as a configuration according to the user manual, while the O_2 XSchema (which might change over time) eliminates some valid configurations that exhibit poor performance. In the manner of BCFG2 or LCFG, information in an XML configuration file (conforming to the XSchema) corresponds with information stored in package configuration files, wherever they might be located. We do not

endorse the use of XML and XSchema for this purpose; we merely remark that they seem powerful enough to serve as a modeling language for this purpose.

To make closure operators practical, we need some form of "strategic schema" for each managed package or unit. This schema serves as a trigger for "tactics" that enforce *one* way to obey the schema. The true purpose of the schema, however, is to allow an operator to leave "well enough" alone. By defining a notion of "health" that is *independent* of the mechanism of assurance, one admits other potentially clever assurance mechanisms that arise from *outside* the operator in question.

7 Properties of Closure Operators

Creating schemas for common packages (or perhaps one might say, "frequently encountered intents") seems a daunting task. Why would one want to do this? In this section, we discuss some of the properties of closure operators as motivation for the work ahead.

First we describe several "composition theorems", that show when a set of closure operators $\mathcal{O} = \{O_1, \ldots, O_k\}$ can be thought of as a single closure operator $\mathcal{O}$, where applying $\mathcal{O}$ consists of randomly choosing and applying one of $O_1, \ldots, O_k$.

First,

Theorem 1. *Suppose we have a set of k closure operators $\mathcal{O} = \{O_1, \ldots, O_k\}$ with the same acceptance set O^{i}. Then the set of operators, viewed as one operator, is a closure operator.*

Proof. View the set of operators as one operator that randomly chooses which sub-operator to call. The acceptance set of this operator is O^{i}, while the assurance set is the *union* of the individual assurance sets O_j^{a}. □

This is just tactic composition in the presence of shared strategy. The most important property of closure operators – as compared to regular Cfengine operators – is *flexibility of response.* Most current operators correspond to "one tactic" for assuring a behavior. "Multiple tactics" can be tried by one closure operator to achieve a coherent aim.

Moreover,

Theorem 2. *Suppose the acceptance sets of $O_1, \ldots, O_k$ intersect in some nonempty set O^{i}. If O contains the assurance set of* one *operator O_j, then the set of these operators, viewed as one random operator, is a closure operator.*

Proof. As before, consider the operator O that randomly calls some O_m when invoked. When O_j is invoked, it assures a state inside the acceptance set. Since operators are chosen randomly to be invoked, it is only a matter of time until it *is* invoked, and other operators in $\mathcal{O}$, once it assures an appropriate state, will not modify O_j's tactics. □

Agreement is not always necessary; orthogonality is sufficient:

Theorem 3. *Suppose that two operators O_1, O_2 operate on orthogonal regions S_1, S_2 of a product space $S_1 \otimes S_2$, so that O_1 affects the chosen subset of S_1 and O_2 affects S_2 only. Then $\{O_1, O_2\}$ is a closure operator.*

Proof. Suppose O_1 assures S_1^{a}, a subset of its acceptance set S_1^{i}, while O_2 assures S_2^{a}, a subset of its acceptance set S_2^{i}. Then $\{O_1, O_2\}$ assures $S_1^{\mathrm{a}} \otimes S_2^{\mathrm{a}}$ and accepts $S_1^{\mathrm{i}} \otimes S_2^{\mathrm{i}}$, so it is a closure operator. □

In other words, closure operators acting on independent entities can be composed into one closure operator.

In general,

Theorem 4. *Suppose that a set of operators $\mathcal{O} = \{O_1, \ldots, O_k\}$ can be factored into subsets of operators, each of which acts on an orthogonal part of configuration, where each subset satisfies the conditions of Theorem 2. Then $\mathcal{O}$ is a closure operator.*

Proof. Apply Theorem 2 to infer that each subset is a closure operator, and then use Theorem 3 inductively to get the result. □

Conversely,

Theorem 5. *Any set of operators that is reachably consistent can be considered as a single closure operator.*

Proof. If a set of operators is reachably consistent, then it shares an acceptance set that is the intersection of all operators' acceptance sets, and for any initial state, the operators achieve some element of that subset, so that at least one operator in the set must have each state in the intersection in its assurance set. □

We have shown, thus, that under a variety of conditions, closure operators compose to make a single closure operator. What, then, is inconsistency? We take the approach of [16], that inconsistency indicates that the operators in a set represent two or more distinct strategies, so that sets of operators can be factored into strategic groups.

Two operators O_1 and O_2 are inconsistent if there is some state in O_1's assurance set that is not in O_2's acceptance set, and vice-versa. It is necessary for lack of acceptance to be symmetric, or the operators would settle among themselves upon the more stringent acceptance set and become consistent. Thus inconsistency is a "flip-flop" situation in which two operators feel compelled to undo each other's tactics.

Detecting inconsistency is difficult if not impossible for distributed operators that do not necessarily have knowledge of each others' strategies and/or goals. We study this problem in [12] and conclude that consistency is best considered to be a *statistical* rather than logical property. We demonstrate methods for evaluating the *hypothesis* that a set of operators is consistent, and relate probability of consistency to time of observation.

8 Conclusions

So far, autonomic computing has been asserted by hierarchical means, in which more limited control loops are part of a larger hierarchy of control. In this paper, we propose a boldly different strategy, of composing control loops as *peers* in a control strategy. In this paper, we have explored the structure of such a set of operators, from a practical and an algebraic standpoint. In a second paper[12], we explore notions of operator consistency that are meaningful and useful in this context.

Closure operators are difficult to build, but have some really nice properties. The most significant of these is that they can be composed with other closure operators without tactical agreement, and the results can emerge as a new control strategy that is the composition of several less-general control strategies. It can be extremely difficult to extend a given operator to create a new function, while that function may be added by another consistent operator with less effort, provided that the two operators share the same basic behavioral model. The behavioral model is thus the pivot upon which effective composition is based, and is needed whether composition is desired or not. The result is a compositional model of autonomics in which operators compose simply because they agree on strategy, but not necessarily on tactics.

There are several directions for future work. First, the theory suggests an extension to Cfengine that allows assurance sets to be smaller than acceptance sets. This is a task of significant complexity, however, because Cfengine currently lacks the modeling machinery required to define acceptance as a concept separate from assurance. Using this extension, we can develop practical examples of cooperative management, such as performance tuning. Aside from inspiring new capabilities for Cfengine, this theory allows other autonomic control loops to be composed in like manner – as peers in an operator calculus. It remains to be seen whether this is better, worse, or just different than composing control loops via hierarchy.

But the theory is most powerful in that it offers a method for dealing with open management situations in which there is no way to establish sufficient closed control loops. By expressing management tools and user actions (including intrusions) as operators, we have one coherent theoretical model for everything that can happen in a network. The future promise of closure operators is that we can express mitigating influences for network problems in terms of closure operator activations and deactivations, rather than in terms of selecting features in a monolithic management tool. This enables highly dynamic management strategies, including intelligent operators whose nature evolves with changing conditions, and that can be deployed, erased, recompiled, and redeployed in a live environment. We believe this high level of dynamism will be required to deal with the ubiquitous computing networks of tomorrow.

9 Acknowledgements

This paper would not exist without the inspiration of the Cfengine community, including Mark Burgess and the many users of Cfengine who have dared to learn a different way of thinking.

References

1. Burgess, M.: A site configuration engine. Computing Systems **8**(2) (1995) 309–337
2. Burgess, M., Ralston, R.: Distributed resource administration using cfengine. Softw., Pract. Exper. **27**(9) (1997) 1083–1101
3. Burgess, M.: Computer immunology. In: LISA, USENIX (1998) 283–298
4. Burgess, M.: Theoretical system administration. In: LISA, USENIX (2000) 1–13
5. Burgess, M.: Cfengine as a component of computer immune-systems. Proceedings of the Norwegian Conference on Informatics (1998)
6. Burgess, M., .Frisch, A.: Promises and cfengine: A working specification for cfengine 3. Technical report, Oslo University College (November 2005)
7. Burgess, M.: An approach to understanding policy based on autonomy and voluntary cooperation. In Schönwälder, J., Serrat, J., eds.: DSOM. Volume 3775 of Lecture Notes in Computer Science., Springer (2005) 97–108
8. Burgess, M., Couch, A.: Autonomic computing approximated by fixed-point promises. In: Proceedings of the First IEEE International Workshop on Modeling Autonomic Communication Environments (MACE), Multicon Verlag (2006) 197–222
9. Couch, A., Hart, J., Idhaw, E.G., Kallas, D.: Seeking closure in an open world: A behavioral agent approach to configuration management. In: LISA '03: Proceedings of the 17th USENIX conference on System administration, Berkeley, CA, USA, USENIX (2003) 125–148
10. Schwartzberg, S., Couch, A.: Experience implementing a web service closure. In: LISA '04: Proceedings of the 18th USENIX conference on System administration, Berkeley, CA, USA, USENIX (2004) 213–230
11. Wu, N., Couch, A.: Experience implementing an ip address closure. In: LISA '06: Proceedings of the 20th USENIX conference on System administration, Berkeley, CA, USA, USENIX (2006) 119–130
12. Couch, A., Chiarini, M.: Dynamic consistency analysis for convergent operators. In: AIMS. (2008) (submitted)
13. Burgess, M., Couch, A.L.: Modeling next generation configuration management tools. In: LISA, USENIX (2006) 131–147
14. Anderson, P.: Configuration Management. SAGE Short Topics in System Administration. USENIX (2007)
15. Couch, A.: Configuration management. In Bergstra, J., Burgess, M., eds.: Handbook of Network and System Administration. Elsevier, Inc. (2007) 75–133
16. Couch, A., Sun, Y.: On the algebraic structure of convergence. In: Proc. DSOM 2003, Springer Berlin (2003) 28–40