Konstruktion selbst-organisierender Softwaresysteme

Hella Seebach

Institut für Software und Systems Engineering, Universität Augsburg
seebach@informatik.uni-augsburg.de

Abstract: Selbst-Organisation ist die Fähigkeit eines Systems, ohne externe Eingriffe die eigene Struktur zu verändern. Sie ermöglicht es somit, dass ein System selbständig sowohl auf interne Veränderungen als auch auf Umweltveränderungen reagieren kann. Dieses Phänomen wird in vielen Disziplinen wie der Physik, der Philosophie und der Biologie untersucht. In der Informatik wurde die Forschung in Richtung dieser Eigenschaft ebenfalls intensiviert. Es wird analysiert, wie Selbst-Organisationsmechanismen, die in der Natur beobachtbar sind, auf Informatiksysteme übertragen werden können. Das Ziel ist es, die steigende Komplexität der Computersysteme in den Griff zu bekommen und die Systeme robuster gegenüber Veränderungen der Umwelt zu machen. In dieser Dissertation wurde ein Verfahren entwickelt, mit dem solche selbst-organisierenden Softwaresysteme top-down, standardisiert und reproduzierbar konstruiert werden können. Eine der größten Herausforderungen war dabei, mit dem nicht vorhersagbaren Verhalten der Systeme umzugehen.

1 Einführung

Die Anforderungen an Softwaresysteme der Zukunft nehmen rasant zu. Vor allem Verteiltheit, Flexibilität und Wiederverwendbarkeit werden gefordert. Zudem sind kurze Entwicklungszeiten wünschenswert. All diese Forderungen sind notwendig, um mit den ständigen Technologieentwicklungen und Anwenderanforderungen, die im Bereich der Informatik aufkommen, mithalten zu können. Die Leistungsfähigkeit und Vernetzung der Rechner steigt stetig und sie können immer komplexere Dienste übernehmen. Es sind somit die technischen Voraussetzungen für flexible, verteilte, vielseitige Systeme vorhanden. Es ist an der Zeit, dass die Softwaretechnik diese Entwicklungen annimmt und Techniken zur Verfügung stellt, mit dieser Komplexität umzugehen und um in diesem Bereich zuverlässige Software standardisiert zu entwickeln.

Die Nutzung der beschriebenen Möglichkeiten macht die Softwaresysteme jedoch noch komplexer und für den Benutzer schwer verwalt- und nachvollziehbar. Um diesen Problemen zu begegnen, entstanden die Forschungsgebiete *Autonomic Computing* [KC03] und *Organic Computing* [Sch05]. In den Bereich des Organic Computing ist diese Dissertation einzuordnen. Organic Computing geht im Vergleich zu Autonomic Computing über Serverarchitekturen hinaus und befasst sich generell mit technischen Systemen, die anhand lokaler Regeln zu einem selbst-organisierenden Gesamtverhalten führen, das „lebensähnlich“ (organisch) wirkt [MS04]. Das Verhalten eines selbst-organisierenden Systems zeigt oft sehr gute Eigenschaften bezüglich Skalierbarkeit und Robustheit gegenüber

Störeinflüssen oder Parameteränderungen, weshalb sich selbst-organisierende Systeme gut als Paradigma für zukünftige, komplexe technische Systeme eignen. Das Verhalten des Gesamtsystem ist im Allgemeinen nicht direkt aus den lokalen Regeln ableitbar. Problematisch ist dabei, dass lokale Regeln nicht unmittelbar nur zu positivem globalem Verhalten führen. Deswegen werden nicht nur Techniken zum Bau solcher Systeme benötigt, sondern auch Techniken, um „schlechtes“ globales Verhalten zu vermeiden.

Im Schwerpunktprogramm „Organic Computing“ der Deutschen Forschungsgemeinschaft [For10] wurden unter anderem Observer-/Controller-Architekturen [RMB+06], und Kooperationsmechanismen [PMGT08] für selbst-organisierende Systeme entwickelt, die ein allgemeines Vorgehen bei der Entwicklung selbst-organisierender Systeme vorschlagen. Trotz dieser generellen Ideen für Organic Computing Systeme befasst sich bisher kaum jemand mit der Fragestellung, wie solche hoch dynamischen Systeme durchgängig von Konzeptmodell bis hin zu Code modelliert und letztendlich konstruiert werden können. Genau diese Vorgehensweise ist jedoch notwendig, um trotz Selbst-Organisation und den damit unvorhersagbaren Verhaltensänderungen eine Akzeptanz der Systeme zu erreichen. Traditionelle Softwaretechnik kommt bei diesen komplexen selbst-organisierenden Systemen an ihre Grenzen [DMSGK06]. So werden beispielsweise Techniken benötigt, um die Flexibilität in die Systeme zu integrieren oder um den Systemen zur Laufzeit Entscheidungshilfen zu geben, damit diese sich „positiv/gewünscht“ verhalten.

Das Ziel dieser Arbeit war es, einen Ansatz für eine ganze Klasse von Systemen zu entwickeln, mit dem durchgängig und „top-down“ solche selbst-organisierenden, zuverlässigen Systeme entwickelt werden können. Mit so einem Vorgehen ist es möglich, die Systeme auf einer formalen Grundlage aufzusetzen und Verhaltensgarantien abzugeben obwohl Selbst-Organisation integriert ist. Leider kann es keinen allumfassenden Ansatz für jede Art von selbst-organisierenden Systemen geben, da diese inhärent zu unterschiedlich sind. Jedoch können die Systeme anhand ihrer Struktur und Eigenschaften klassifiziert werden und anschließend für diese Klassen von Systemen Techniken und Modelle entwickelt werden. Die Dissertation stellt Techniken für die Klasse der Ressourcenflusssysteme vor und gibt einen Ausblick auf weitere Systemklassen, die mit den entwickelten Techniken selbst-organisierend realisiert werden können. Die entwickelte Methodik umfasst mehrere Elemente, die in einer Software Engineering Guideline zusammengefasst werden. Die Guideline gibt dem Entwickler eine Anleitung an die Hand, wann welche Techniken wie anzuwenden sind. Die Techniken reichen von Softwaretechnik und Algorithmen bis hin zu Formalen Methoden, um erwünschtes globales Verhalten zu erreichen und unerwünschtes zu vermeiden.

2 Selbst-Organisation kontrollieren durch Verhaltenskorridore

Selbst-organisierende Systeme sind in der Lage, sich selber neu zu konfigurieren und sich im Falle von Änderungen ihres Zustandes (z.B. Ausfall einer Fähigkeit) bzw. der Umgebung an diese anzupassen. Diese Fähigkeiten der Systeme machen sie im Vergleich zu traditionellen, robuster, zuverlässiger und auch leichter zu warten, da kein menschliches Eingreifen mehr erforderlich ist. Jedoch stellt sich die Frage, wie sichergestellt werden

kann, dass diese Systeme auch nur das tun, wozu sie entworfen wurden. Der entwickelte *Restore Invariant Approach (RIA)* stellt eine Technik dar, die eine Grundlage für die Konstruktion solcher Systeme mit sogenannten Selbst-X-Eigenschaften (Selbst-Heilung, Selbst-Optimierung, etc.) schafft. Der RIA ist die Schnittstelle, die zentrale Idee, um Softwaretechniken und formale Analyse für selbst-organisierende Systeme auf eine gemeinsame Basis zu stellen und eine einheitliche Zielvorstellung zu entwickeln.

Die Grundidee ist es, dem System einen Verhaltenskorridor vorzugeben, in dem es sich völlig frei bewegen kann. Sobald das System durch Einflüsse wie Ausfälle, neue Komponenten oder neue Aufgaben diesen Korridor verlässt, greift ein Rekonfigurationsmechanismus ein, der das System wieder in den Korridor zurückbringt. Wie auch traditionelle Systeme sind selbst-organisierende Systeme prinzipiell nicht vor Fehlern oder neu auftretenden Ereignissen geschützt. Die Zustände, die ein selbst-organisierendes System durchlaufen kann, können somit unterschieden werden in funktionale Zustände $\mathcal{S}_{func}$, in denen das System seine Aufgabe erfüllt und Zustände $\mathcal{S}_{reconf}$, in denen das System gerade rekonfiguriert, um nach dem Auftreten einer Veränderung, wieder in einen funktionalen Zustand zurückzukehren. Für den Zustandsraum $\mathcal{S}$ des Systems gilt die folgende Bedingung:

$$\mathcal{S} := \mathcal{S}_{func} \cup \mathcal{S}_{reconf} \text{ , mit } \mathcal{S}_{func} \cap \mathcal{S}_{reconf} = \emptyset.$$

Das heißt, ein System ist immer entweder funktionsfähig oder wird gerade rekonfiguriert. Der Korridor zieht somit die Linie zwischen den funktionalen Systemzuständen und den nicht erwünschten Zuständen, in denen rekonfiguriert werden muss.

Formal kann das Aufsplitten der Zustände als prädikatenlogische Formel beschrieben werden, die zu *wahr* ausgewertet wird für die Zustände $\mathcal{S}_{func}$ und zu *falsch* für die Zustände $\mathcal{S}_{reconf}$. Nachdem die Formel für alle funktionalen Zustände wahr ist, wird von einer *Invariante*, die für das System gelten soll, gesprochen. Der Verhaltenskorridor für selbst-organisierende Systeme wird über Bedingungen (Constraints) spezifiziert, d.h. es wird auf den Systemkomponenten definiert, was zulässig ist und was nicht. Die Konjunktion all dieser Constraints bildet die Invariante. Um diese Bedingungen spezifizieren zu können, wird ein Modell des Systems benötigt (s. Kapitel 4). Auf diesem Modell werden die Bedingungen mit Hilfe der Object Constraint Language (OCL) spezifiziert. Diese müssen zur initialen Konfiguration und zur Laufzeit permanent ausgewertet werden. Bei der Verletzung eines Constraints und somit der Invariante, wird der Korridor verlassen und es muss eine neue Belegung für die Zustandsvariablen gefunden werden, so dass die Invariante wieder Gültigkeit hat (restore). Dazu muss die Invariante über Zustandsvariablen sprechen, denen neue Werte zugewiesen werden können. Dies ist der Fall, wenn das Produktivsystem Freiheitsgrade, sprich Redundanz enthält. Der Rekonfigurationsmechanismus muss also für eine Belegung sorgen, die einem Zustand des Produktivsystems entspricht, der innerhalb des Korridors liegt.

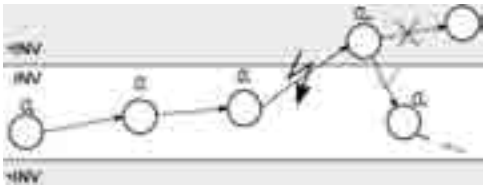

Abbildung 1: Verhaltenskorridor

Abbildung 1 zeigt den Zusammenhang zwischen Systemzuständen, Invariante (INV) und dem Fall, dass in dem Schritt von σ_2 zu σ_{err} das System einen Fehlerzustand betritt, der

außerhalb des Korridors liegt. Durch die Rekonfiguration betritt das System im nächsten Schritt σ_4 wieder den Korridor (mit $\sigma_0, \sigma_1, \sigma_2, \sigma_4 \in \mathcal{S}_{func}$ und $\sigma_{err} \in \mathcal{S}_{reconf}$).

Liegt ein solcher Verhaltenskorridor und eine Systemspezifikation für ein selbst-organisierendes System vor, können Aussagen über das Verhalten des Systems gemacht werden, bzw. über Eigenschaften, die das System innerhalb des Korridors aufweist.

Der RIA ermöglicht es einem Systementwickler, den funktionalen Teil des Systems herkömmlich zu beschreiben und zu modellieren und den Rekonfigurationsteil völlig separat zu betrachten. Der Teil, der für die Rekonfiguration zuständig ist, muss lediglich die Aufgabe erfüllen, Invarianten wiederherzustellen. Die Berechnung korrekter Variablenbelegungen kann als „Constraint Satisfaction Problem“ beschrieben werden. Wie dies konkret gelöst wird, bleibt dem Rekonfigurationsmechanismus überlassen. Beispiele für Rekonfigurationsmechanismen (s. Kapitel 5) sind Constraint Solver oder genetische Algorithmen. Ebenso gibt die Theorie des RIA nicht vor, dass diese Invariante systemweit ausgewertet werden muss. In den meisten Fällen können die Bedingungen lokal überwacht und ausgewertet werden. Dies erlaubt lokale, also dezentrale Rekonfiguration.

Wie bereits erwähnt ist es nicht möglich, diesen Verhaltenskorridor für jegliche selbstorganisierende Systeme einmalig zu spezifizieren. Vielmehr ist es die Aufgabe Systemklassen zu finden, die modelliert werden und auf deren Modellen dieser Korridor definiert werden kann. In der Dissertation wurde die Systemklasse der Ressourcenflusssysteme eingehend untersucht und modelliert. Im Folgenden wird eine Beispielanwendung (s. Kapitel 3) aus dieser Systemklasse vorgestellt und anschließend das Organic Design Pattern (s. Kapitel 4), welches alle notwendigen Komponenten und deren Verhalten beschreibt. Auf diesen Komponenten wurden alle Constraints definiert, die notwendig sind, um den Verhaltenskorridor für Ressourcenflusssysteme im Generellen zu spezifizieren.

3 Fallstudie: adaptive Produktionszelle

Inspiriert von den Problemstellungen die herkömmliche Produktionsstraßen aufweisen (starr, unflexibel, hoch spezialisiert [HNS+10]) und den Bestrebungen, diese flexibler zu gestalten [BS07], ist die Anwendung der adaptiven Produktionszelle entstanden. Selbst-Organisation ist ein Mittel um diese Systeme einerseits dezentral und ausreichend flexibel, andererseits beweisbar zuverlässig zu konstruieren. Die adaptive Produktionszelle beinhaltet Roboter und mobile Plattformen (Carts), die sich an wechselnde Bedingungen anpassen können. Das heißt, dass Roboter verschiedene Fähigkeiten haben (Werkzeugwechsel) und Carts so mobil sind, dass sie Ressourcen zwischen beliebigen Robotern transportie-

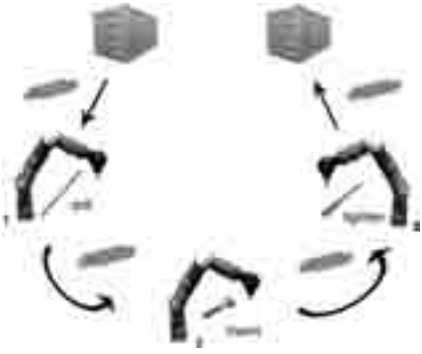

Abbildung 2: Adaptive Produktionszelle

ren können. Eine initiale Konfiguration, wie sie in Abbildung 2 gezeigt wird, ist eine Möglichkeit so eine Produktionszelle aufzubauen. Jeder Roboter verfügt über drei Fähigkeiten (drill, insert, tighten). Die Idee der adaptiven Produktionszelle ist es, die eingeführten Freiheitsgrade zu nutzen, um auf Ausfälle und neue Aufgaben, die dem System gestellt werden, reagieren zu können. Gerade für Kleinserien ist die adaptive Produktionszelle gut geeignet. Durch die Flexibilität in den Transportwegen, können die Roboter in beinahe beliebiger Reihenfolge angesteuert werden und somit kann ohne große Umbauarbeiten eine neue Aufgabe erfüllt werden. Des Weiteren stellte ein fixes Förderband bisher einen *Single Point of Failure* dar, d.h. wenn das Band stehen blieb, musste die gesamte Produktionsstraße angehalten werden.

4 Definition selbst-organisierender Ressourcenflusssysteme

Das Organic Design Pattern (ODP) [SOR07] ist ein Werkzeug, das die Konstruktion von selbst-organisierenden Systemen erleichtert und formale Aussagen über Systemeigenschaften ermöglicht. Das Herzstück des Patterns ist ein erprobtes Rollenkonzept, das es erlaubt, Systeme zu modellieren, die sich selbständig an neue Aufgaben anpassen und verschiedene Ressourcen mit verschiedenen Aufgaben gleichzeitig bearbeiten. Die im ODP vordefinierten Constraints spezifizieren den angesprochenen Verhaltenskorridor und auch dessen Überwachung zur Laufzeit. Des Weiteren gibt das Pattern das Verhalten der funktionalen Teile des Systems bereits vor. In [NSS+10] wird gezeigt, wie dieses Verhalten verifiziert werden kann. Unter der Annahme, dass der Rekonfigurationsmechanismus ausschließlich die Konfiguration beeinflusst und nicht in das Verhalten der funktionalen Komponenten eingreift, ist somit verifiziert, dass das selbst-organisierende System funktional korrekt arbeitet. Durch einen verifizierten Result Checker [FNSR11] wird sichergestellt, dass die Konfigurationen, die von dem Rekonfigurationsmechanismus berechnet wurden, korrekt sind bzgl. der Einhaltung aller Constraints.

Die statischen Elemente eines ODP-Systems und deren Zusammenhänge sind in Abbildung 3 als UML Klassendiagramm aufgezeigt. Das wichtigste Konzept ist der *Agent*, der entsprechend einer Aufgabe (*Task*) die *Resource* mit einer oder mehreren Fähigkeiten (*Capabilities*) bearbeitet. Der Agent kann zu jedem Zeitpunkt maximal eine Ressource bearbeiten. Der Task beschreibt eine Abfolge von Capabilities (ordered, nonunique), die auf eine Ressource angewendet werden sollen. Er ändert sich während der Verarbeitung einer Ressource nicht. Der Zustand (*state*) der Ressource ist immer ein Präfix des Tasks. Agenten, die eine Ressource produzieren, stellen den Startpunkt des Ressourcenflusses dar, Agenten, die eine Ressource konsumieren, entsprechend den Endpunkt. Jeder Agent zeichnet sich durch die Fähigkeiten, die er besitzt (*availableCapabilities*) und seine Interaktionsmöglichkeiten aus. Letztere geben an, welchen anderen Agenten der jeweilige Agent Ressourcen geben (*outputs*) kann und von welchen anderen Agenten er Ressourcen annehmen (*inputs*) kann. Die Agenten verfügen nur über ihr lokales Wissen, d.h. sie kennen im Allgemeinen nur die Agenten, die ihnen über genau diese „Input-/Output-Relationen“ bekannt sind. Die Kommunikation in ODP-Systemen ist nicht eingeschränkt, d.h. sobald ein Agent einen anderen Agenten „kennt“, kann er mit diesem kommunizieren.

Ein Agent kann mehrere *Rollen* haben. Die Zuweisung von Rollen zu Agenten wird Rollenallokation genannt (*allocatedRoles*). Selbst-Organisation wird in dieser Systemklasse als ein Rollenallokationsproblem angesehen: Welcher Agent muss zu welchem Zeitpunkt, welche Rolle ausführen, damit das Gesamtsystem korrekt arbeitet? Die momentan vom Agenten gewählte Rolle bestimmt, welche Fähigkeiten der Agent auf die vorliegende Ressource anwendet. Eine Rolle setzt sich aus drei Teilen zusammen, einer Vorbedingung (*precondition*), einer Sequenz von Fähigkeiten und einer Nachbedingung (*postcondition*). Die Vorbedingung gibt an, von welchem Agenten (*port*) der Agent die Ressource bekommt, welchen Task die Ressource hat und in welchem Bearbeitungszustand (*state*) sie momentan ist. Die Sequenz von Fähigkeiten (*capabilitiesToApply*), die in der Rolle angegeben sind, bestimmen, was der Agent konkret mit der Ressource zu tun hat. Das kann von „nur durchreichen“ (leere Sequenz) bis „alles erledigen“ (alle Capabilities des Tasks) reichen. Die Nachbedingung gibt wiederum an, mit welchem Zustand die Ressource den Agent verlässt, welche Aufgabe die Ressource hat und zu welchem Agent (*port*) die Ressource weitergegeben werden soll. In dem Beispiel der adaptiven Produktionszelle (s. Abbildung 2) nimmt Roboter 1 die Ressource von Cart 1 in dem Zustand [], mit dem Task [$drill$, $insert$, $tighten$], führt $drill$ aus und gibt die Ressource weiter an Cart 2 mit dem Zustand [$drill$].

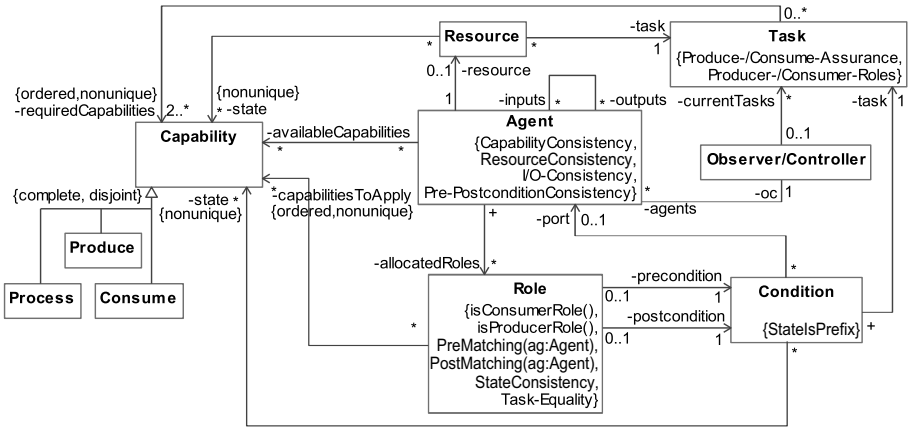

Abbildung 3: Organic Design Pattern - Konstruktionsmodell

Wie in Kapitel 2 beschrieben, werden eine Reihe von Constraints benötigt, die auf den Komponenten des Zielsystems definiert werden können, um Verhaltensgarantien abzugeben. Die hier vorgestellte statische Struktur bildet nun die Grundlage, um mit OCL-Constraints [OMG06] einen Verhaltenskorridor zu spezifizieren. Die Constraints sind in abgekürzter Schreibweise in den Komponenten des ODP zu finden. An dieser Stelle wird exemplarisch ein Constraint angegeben. Der *CapabilityConsistency*-Constraint (vgl. Listing 1) eines Agenten besagt, dass ein Agent nur Rollen zugewiesen bekommen darf, in denen er Capabilities anwenden muss, die er in seiner *availableCapabilities*-Relation hat.

Diese Constraints werden auf der einen Seite benötigt, um eine korrekte Rollenallokation zu berechnen und auf der anderen Seite, um durch Beobachtung eine Verletzung der

Invariante festzustellen. Im gegebenen Beispiel können die Agenten (Roboter und Carts) direkt feststellen, ob ihre Rollenallokation noch korrekt ist und ggf. eine Verletzung des Constraints und somit der Invariante melden bzw. lokal verarbeiten. Fällt nun bei einem Roboter der Bohrer *drill* aus muss er mit einem anderen Roboter die Rolle tauschen. Die Carts rekonfigurieren ihre Transportrollen entsprechend und stellen somit wieder einen korrekten Ressourcenfluss her. Für diese Rollenrekonfiguration muss genügend Redundanz im System vorhanden sein, sowohl in den Capabilities als auch in den Input-Output Relationen.

```
1 context Agent inv: availableCapabilities ->
2   includesAll(allocatedRoles.capabilitiesToApply)
```

Listing 1: CapabilityConsistency

Der konkrete Rekonfigurationsmechanismus, der das System nach Verlassen des Korridors wieder in diesen zurück bringt, ist gekapselt in der Observer/Controller-Komponente (O/C). Das Ergebnis des Rekonfigurationsalgorithmus ist eine neue Allokation von Rollen zu Agenten, die das Gesamtsystem in einen Zustand bringt, in dem es konform zu den Constraints arbeitet und wieder in der Lage ist, das Systemziel korrekt zu erfüllen. Die Constraints und statischen Komponenten des ODP bestimmen die Ergebnisform des Rekonfigurationsalgorithmus. Die Klassen, Relationen und Kardinalitäten gekoppelt mit den Constraints spezifizieren korrekte Resultate und somit welche Rollenallokationen für Systeme, die mit dem ODP modelliert wurden, gültig sind. Ein Zustand eines ODP-Systems ist eine konkrete Instanz des ODP. Die Zustände σ_0-σ_4 aus Abbildung 1 entsprechen jeweils einer korrekt konfigurierten Instanz des ODP. Der Schritt zwischen diesen Zuständen entspricht einer Änderung der ausgehenden Instanz, z.B. durch neue Objekte oder veränderte Assoziationen. $\sigma_{err} \in \mathcal{S}_{reconf}$ entspricht einer Instanz, die im Moment nicht korrekt konfiguriert ist, da z.B. in dem Schritt, der zu diesem Zustand führte, ein Agent die Fähigkeit, die er für seine Rolle benötigt, verloren hat. Die statischen und dynamischen Konzepte des ODP sind in der Referenzimplementierung *ODP Runtime Environment (ORE)* komplett umgesetzt. Die Implementierung setzt auf Jadex, einem Multi-Agenten-System auf und unterstützt eine einfache Konstruktion selbst-organisierender Ressourcenflusssysteme.

5 Rekonfiguration selbst-organisierender Ressourcenflusssysteme

Detektiert nun ein Agent lokal eine Constraintverletzung, muss ein Teil des Systems rekonfiguriert werden. Um diese Rekonfiguration durchzuführen bietet die vorliegende Arbeit prinzipiell zwei verschiedene Möglichkeiten an. Zum einen kann das gesamte System durch eine zentrale Instanz rekonfiguriert werden, die das Wissen über das Gesamtsystem hält. Die zweite Möglichkeit ist eine dezentrale Rekonfiguration die nur einen kleinen Teil des Systems mit neuen Rollen ausstattet. Eine zentrale Instanz zieht immer einige Nachteile mit sich, die hier nicht im Detail erläutert werden sollen. Vielmehr wird im Folgenden ein kurzer Einblick in die dezentrale Rekonfiguration gegeben, die minimale Eingriffe in das System ermöglicht und Entscheidungen auf lokalem Wissen fällt. Somit stellt nach [DMSGK05] die dezentrale Rekonfiguration einen stark selbst-organisierenden Mechanismus dar.

Die Grundidee der dezentralen Rekonfiguration [ASN+11] ist die Bildung von Agentengruppen, *Koalitionen* genannt, die in der Lage sind, verletzte Constraints wieder zu erfüllen. Fehler, die durch den Ausfall von Capabilities, Input- oder Output-Ports sowie vollständiger Agenten entstehen, werden kompensiert, indem für einen Teil des Systems neue Rollen berechnet werden. Die anderen Bereiche des Systems, die an der Rekonfiguration nicht beteiligt sind, können für den betroffenen Task weiterhin die alloziierten Rollen ausführen. Dabei existieren die Koalitionen ausschließlich in Phasen der Rekonfiguration. Sind diese abgeschlossen, so lösen sich die Koalitionen wieder auf. Dieser Mechanismus profitiert von der festgelegten Systemstruktur durch das ODP und kann somit anhand lokaler Regeln über die Gültigkeit der Invariante Entscheidungen treffen.

Diese dezentrale Kontrolle bietet die Möglichkeit nur sehr selektiv in den laufenden Betrieb der Systeme einzugreifen. Es werden nur die Agenten angehalten, bei denen eine Constraintverletzung vorliegt, sowie angrenzende Agenten, die bei der Wiederherstellung einer korrekten Rollenallokation helfen können. Das Ziel ist es, effizient und effektiv den Ressourcenfluss wiederherzustellen. In Systemen, die mit dem ODP konstruiert wurden und die den Koalitionsbildungsmechanismus zur Rekonfiguration nutzen, ist jeder einzelne Agent in der Lage, eine Koalition zu bilden, um das System wieder in einen Zustand innerhalb des Korridors zurückzuführen. Dieser Mechanismus greift somit nicht in die Eigenständigkeit der Agenten ein, was im Sinne der Selbst-Organisation natürlich wünschenswert ist. Wenn die Entscheidung für die Rekonfiguration auf einen dezentralen Mechanismus fällt, muss jedoch in Kauf genommen werden, dass dieser Mechanismus nicht immer die global optimale Lösung finden kann, speziell wenn er auf rein lokalem Wissen arbeitet. Ein Vorteil des in dieser Dissertation präsentierten Gesamtansatzes ist es, dass im ORE verschiedene Rekonfigurationsmechanismen für die Realisierung starker (dezentraler) sowie schwacher (zentraler) selbst-organisierender Systeme bereitgestellt werden. Der Entwickler eines selbst-organisierenden Ressourcenflusssystems ist damit in der Lage, die für seine Anwendung optimale Rekonfigurationsstrategie mittels eines Plugin-Mechanismus einzusetzen.

6 Innovationen und Ausblick

Motiviert von der zunehmenden Komplexität zukünftiger Softwaresysteme ist es in dieser Dissertation gelungen, einen Ansatz zu entwickeln, der das Design und die Konstruktion selbst-organisierender Systeme erleichtert, mit denen die steigende Komplexität handhabbar wird. Wesentliche Ergebnisse dieser Arbeit sind der *Restore Invariant Approach*, der die Definition eines Korridor gewünschten Verhaltens fordert und die getrennte Betrachtung des funktionalen Systems und den Mechanismen der Selbst-Organisation ermöglicht; das *Organic Design Pattern*, welches die Klasse der selbst-organisierenden Ressourcenflusssysteme beschreibt; verschiedenste Rekonfigurationsmechanismen, sowohl zentraler als auch dezentraler Natur; das *ODP Runtime Environment*, das alle entwickelten Mechanismen in einem Framework vereint und die *Software Engineering Guideline* [SNSR10], die letztendlich eine Anleitung zur einfachen, reproduzierbaren Entwicklung selbst-organisierender Systeme darstellt. Auf einige dieser Punkte konnte leider im Rahmen die-

ses Beitrags nicht eingegangen werden, ausführliche Beschreibungen finden sie jedoch in [See11].

Aus den Ergebnissen der Dissertation ergeben sich einige neue Möglichkeiten und interessante Fragen. Die Spezifikation der korrekten Systemkonfiguration basiert bereits auf OCL, eine weiterführende spannende Frage ist, was mit dieser Spezifikation noch auf Modellebene ausgedrückt werden kann. Interessant ist dabei die Analyse eines konkreten Systems (d.h. einer Instanz des ODP), bzgl. der Redundanzverteilung. Es könnte getestet werden, ob das System optimal konfiguriert ist, damit der Selbst-Organisationsmechanismus die Zuverlässigkeit und Robustheit des Systems wesentlich verbessern kann. Untersucht werden kann diese Instanz unter anderem auch auf das Vorkommen von *Single Point of Failures*. Solche Schwachstellen bereits auf Modellebene zu finden ist eine erstrebenswerte Erweiterung des bisherigen Ansatzes. Eine weitere Herausforderung ist es, die Erfahrung mit dem Design selbst-organisierender Systeme zu nutzen, um das ODP auch für andere Systemklassen nutzbar zu machen. Interessant sind dabei die folgenden Fragestellungen: welche Konzepte des ODP werden für welche Systemklasse benötigt; werden manche Teile eventuell nicht benötigt; müssen neue Konzepte und Constraints definiert werden; wie gliedern sich diese Konzepte ein; welche Abhängigkeiten bestehen unter den einzelnen Konzepten und den Constraints, die über diese Konzepte sprechen. Die Vision ist, dass nach einer ausgiebigen Analyse weiterer Systemklassen eine Art Baukasten für selbst-organisierende Systeme entsteht.

Literatur

[ASN+11] G. Anders, H. Seebach, F. Nafz, J.-P. Steghöfer und W. Reif. Decentralized Reconfiguration for Self-Organizing Resource-Flow Systems Based on Local Knowledge. In *Proceedings of EASe 2011*, Las Vegas, Nevada, USA, April 2011.

[BS07] B. Bickel und M. Schuster. *Trends, Methoden und Grundsätze moderner Fabrik- und Produktionsplanung*. GRIN Verlag, 2007.

[DMSGK05] G. Di Marzo Serugendo, M.-P. Gleizes und A. Karageorgos. Self-organization in multi-agent systems. *The Knowledge Engineering Review*, 20(02):165–189, 2005.

[DMSGK06] G. Di Marzo Serugendo, M.P. Gleizes und A. Karageorgos. Self-Organisation and Emergence in MAS: An Overview. *Informatica*, 30(1):45–54, 2006.

[FNSR11] P. Fischer, F. Nafz, H. Seebach und W. Reif. Ensuring Correct Self-Reconfiguration in Safety-Critical Applications by Verified Result Checking. In *Workshop Organic Computing as part of ICAC 2011*, Karlsruhe, Germany, June 2011.

[For10] Deutsche Forschungsgemeinschaft. http://www.organic-computing.de/SPP, 2010. DFG SPP 1183 - Organic Computing Initiative.

[HNS+10] A. Hoffmann, F. Nafz, H. Seebach, A. Schierl und W. Reif. Developing Self-Organizing Robotic Cells using Organic Computing Principles. In *Workshop on Bio-Inspired Self-Organizing Robotic Systems, 2010 IEEE International Conference on Robotics and Automation (ICRA 2010)*. Springer, 2010.

[KC03] J.O. Kephart und D.M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[MS04] C. Müller-Schloer. Organic Computing - On the Feasibility of Controlled Emergence. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/-software codesign and system synthesis*, Seiten 2–5. ACM, 2004.

[NSS+10] F. Nafz, H. Seebach, J.-P. Steghöfer, S. Bäumler und W. Reif. A Formal Framework for Compositional Verification of Organic Computing Systems. In *Proceedings of the 7th International Conference on Autonomic and Trusted Computing*. Springer, 2010.

[OMG06] OMG. Object Constraint Language, OMG Available Specification, 2006. Version 2.0.

[PMGT08] H. Parzyjegla, Jaeger M., Mühl G. und Weis T. Model-driven Development and Adaptation of Autonomous Control Applications. *IEEE Distributed Systems Online (DSOnline)*, 9(11), November 2008.

[RMB+06] U. Richter, M. Mnif, J. Branke, C. Müller-Schloer und H. Schmeck. Towards a generic observer/controller architecture for Organic Computing. *INFORMATIK 2006 – Informatik für Menschen!*, P-93:112 – 119, 2006.

[Sch05] H. Schmeck. Organic Computing - A New Vision for Distributed Embedded Systems. In *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC '05, Seiten 201–203, Washington, DC, USA, 2005. IEEE Computer Society.

[See11] H. Seebach. *Konstruktion selbst-organisierender Softwaresysteme*. Dissertation, University of Augsburg, Augsburg, Germany, 2011. (in German).

[SNSR10] H. Seebach, F. Nafz, J.-P. Steghöfer und W. Reif. A Software Engineering Guideline for Self-organizing Resource-Flow Systems. In *Proceedings of the 4th IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, 2010.

[SOR07] H. Seebach, F. Ortmeier und W. Reif. Design and Construction of Organic Computing Systems. In *Proceedings of the IEEE Congress on Evolutionary Computation*, 2007.

Hella Seebach wurde 1978 in Oldenburg geboren und hat Informatik an der Universität Augsburg studiert. Sie promovierte am Lehrstuhl „Softwaretechnik und Programmiersprachen“ bei Professor Reif. Zur Zeit koordiniert Frau Seebach dort den Bereich Organic Computing, der das DFG geförderte Projekte SAVE ORCA und die DFG-Forschergruppe OC-Trust umfasst. In beiden Projekten stehen selbst-organisierende Systeme im Mittelpunkt. SAVE ORCA befasst sich mit den generellen Konstruktionsparadigmen und formaler Korrektheit der Systeme, während in OC-Trust der Fokus auf der Entwicklung von Trustmechanismen für selbst-organisierende Systeme liegt.