An Application of Specification-based Design of Self-Stabilization to Tracking in Wireless Sensor Networks

Murat Demirbas[1] and Anish Arora[2]

[1] Computer Science & Engineering Dept.
University at Buffalo, SUNY, Buffalo, NY, 14260
demirbas@cse.buffalo.edu
[2] Computer Science & Engineering Dept.
The Ohio State University, Columbus, OH, 43210
anish@cse.ohio-state.edu

Abstract. In previous work, we have designed a tracking protocol, Stalk, for wireless sensor networks and proved it to be self-stabilizing at the pseudo-code (I/O automata) level. However, it is very challenging to achieve and verify self-stabilization of the same protocol at the implementation (TinyOS) level due to the size of the corresponding program at the implementation level. In this paper, we present a lightweight and practical method for specification-based design of stabilization and illustrate this method on the Stalk protocol as our case study.

1 Introduction

In previous work [1] we presented a self-stabilizing tracking service, Stalk, for sensor networks. There, we used I/O automata specification language for describing Stalk, and gave formal proofs of correctness and self-stabilization for this I/O language program. The implementation languages for sensor network platforms are, however, more finer-grained than the abstract I/O language. For the mote [2] platform, the implementation language is a dialect of C, called NesC [3], and the runtime environment TinyOS [4] consists of a collection of system components for network protocols and sensor drivers. With a conservative estimate, the 20 lines of I/O code we wrote for Stalk will correspond to 2000 lines of code (including the libraries for networking and sensing) at the implementation level. Even though we formally verified Stalk at the I/O language level, proving correctness and self-stabilization of the corresponding implementation at the TinyOS level by studying 2000 lines of code is a very challenging task.

There have been several work on fault-tolerance preserving refinements [5–8]. One can consider using these refinements for implementing Stalk in TinyOS, however, these refinements do not have compiler/code-transformer tool support and, hence, their adoption in practice is limited. In this case, it would be hard to prove manually that our implementation at the TinyOS level is in fact a stabilization-preserving refinement of Stalk at the I/O automata level.

Contributions of this paper: In this paper we present a lightweight and practical method for specification-based design of stabilization. More specifically, we show that we can use ordinary refinements (for which a lot of compiler/tool support exists) and still achieve a specification-based design of stabilization under suitable conditions. We illustrate our lightweight and practical specification-based design method on the Stalk protocol as our case study.

An outline of our lightweight method for specification-based design of stabilization. Given a high-level system specification A, the specification-based approach [5, 6] is to design a tolerance wrapper W such that adding W to A yields a fault-tolerant system. The goal is to ensure that for any low-level implementation C of A adding a low-level implementation W' of W would also yield a fault-tolerant system. Since the refinements from A to C and W to W' can be done independently, specification-based design enables a posteriori or dynamic addition of fault-tolerance. That is, given a concrete implementation C, it is possible to add fault-tolerance to C by first designing an abstract tolerance wrapper W using solely an abstract specification A of C, and then adding a concrete refinement W' of W to C.

We next present a brief outline of our method for adopting ordinary refinements for specification-based design of stabilization in terms of a series of challenges and fixes.

Challenge: Refinements do not preserve fault-tolerance. Ordinary refinements do not preserve fault-tolerance and do not support fault-tolerance composition: Even though the abstract system composed of the fault-intolerant tracking program A and the self-stabilization wrapper W is self-stabilizing, when A and W are refined into C and W' at the implementation level, the concrete system might not be stabilizing since starting from faulty states C may interfere with and invalidate the recovery strategy of W'. Even when one proves that starting from faulty states A does not interfere with W, since ordinary refinements are concerned only with computations starting from good states, computations of C that start from faulty states are unconstrained and may be interfering with W'.

Fix: Use atomic wrappers to avoid interference. In order to prevent the interferences between the wrapper and the application code outside the good states, we use atomic wrappers at both the abstract and the concrete systems. When an atomic wrapper is executed it corrects the application to a good state in a single step, and the application code does not have the opportunity to interfere with the execution and the recovery strategy of the wrapper. Similarly, we also require that the wrapper self-stabilizes atomically in order to prevent the application to interfere with the self-stabilization of the wrapper when starting from a faulty state for the wrapper.

Challenge: Atomic wrappers are infeasible for distributed systems. In a distributed system, global system state is not available for instantaneous access. So, it is unrealistic to assume a wrapper that can in one step correct the entire application state, which is distributed across the system/network.

Fix: Use local atomic wrappers per each process. For effective specification-based design of stabilization of distributed systems, we restrict our attention to wrappers local to each process of the distributed system. At the abstract level, $A = (\Box\ i ::\ A_i)$ [3], we design the wrappers to be decomposable as local wrappers, one for every process i; i.e., $W = (\Box\ i ::\ W_i)$. While refining to a distributed implementation $C = (\Box\ i ::\ C_i)$ we refine these local and atomic wrappers to be composed with the application code C_i at each process i; i.e., $W' = (\Box\ i ::\ W'_i)$

Challenge: Composed system as a whole may still fail to be stabilizing. By using local and atomic wrappers we achieve stabilization for each process both at the abstract and concrete system levels. However, even though all the processes are individually stabilizing, the system may fail to stabilize as a whole due to the continuous introduction of corruptions to the system by the processes that are in a faulty state at the time. Consider a scenario where process j is not yet stabilized but i is. If they interact, i may receive bad input from j, and its state may become bad. Next, when j is corrected to a good state, since i is not yet stabilized, i can in turn infect j. This cycle may repeat infinitely, and even though i and j are individually stabilizing, the system may fail to stabilize as a whole.

Fix: Use a compositional framework. In order to ensure that stabilization of individual processes leads to stabilization of the system as a whole, we borrow ideas from literature on compositional approaches to stabilization. One simple idea is stabilization through composition of layers [10]. In the traditional stabilization by layers approach lower-level processes are oblivious to the existence of higher-level processes, and higher-level processes can read (but not write) the state of a lower-level process. Processes can corrupt each other, but only in a predetermined controlled way since lower-level processes cannot be affected by the state of higher-level ones. Also, the order in which correction must take place is the same direction; the correction of higher-levels depend on that of lower-levels. In order to ensure that stabilization compose at the system level, we adopt a layered composition technique at the abstract system level and assert that the concrete system preserve the layered composition structure of the abstract system.

Application to Stalk. To recap, we make the following assumptions in our method:

1. Wrappers are local to each process and are atomic.
2. Identical layered composition structure is used by the abstract and concrete systems.

[3] A formula $(op\ i : R.i : X.i)$ denotes the value obtained by performing the (commutative and associative) op on the $X.i$ values for all i that satisfy $R.i$. As special cases, where op is conjunction, we write $(\forall i : R.i : X.i)$, and where op is disjunction, we write $(\exists i : R.i : X.i)$. Thus, $(\forall i : R.i : X.i)$ may be read as "if $R.i$ is true then so is $X.i$", and $(\exists i : R.i : X.i)$ may be read as "there exists an i such that both $R.i$ and $X.i$ are true". Where $R.i$ is true, we omit $R.i$. This notation is adopted from [9]. In the above formula, the operator $\Box$ denotes the union of automata/processes.

These two assumptions are satisfied by a rich class of implementations. In our case, Stalk satisfies both assumptions. The correctors (wrappers) in Stalk are local to each process and atomic: the process is atomically put into a locally consistent state with respect to the processes it interacts. Also Stalk algorithm imposes a static structure on the information flow. There is no communication from a higher level process to a lower level process in Stalk. The direction of communication in the protocol is always from lower level processes to higher level processes. Due to this structural constraint, the same layered composition structure is applicable at both the abstract and concrete systems.

We show, using these two assumptions, that an ordinary refinement suffices for the fault-intolerant tracking algorithm, and a self-stabilization preserving refinement suffices for the wrappers. The reason we use a stabilization-preserving refinement for the wrapper is to ensure that the concrete wrapper is able to stabilize from the corruption of its variables. Since there are a lot of tool support for ordinary refinements, refinement of the tracking algorithm can be done automatically via a compiler. Since the wrappers are small and simple their proof of self-stabilization can be achieved easily even at the implementation level.

Outline of the rest of the paper. We present the system model in Section 2. We then prove in Section 3 that the refinement method we described above is amenable for the specification-based design approach. We discuss the refinement of Stalk to a self-stabilizing implementation in Section 4. After presenting the related work in Section 5, we conclude the paper in Section 6.

2 Model

Let Σ be a state space.

Definition. A *system* S is a finite-state automaton (Σ, T, I) where T, the set of transitions, is a subset of $\{(s_0, s_1) : s_0, s_1 \in \Sigma\}$ and I, the set of initial states, is a subset of Σ.

A computation of S is a maximal sequence of states such that every state is related to the subsequent one with a transition in T, i.e., if a computation is finite there are no transitions in T that start at the final state.

We refer to an abstract system as a *specification*, and to a concrete system as an *implementation*. For convenience in this paper we assume that the specification and the implementation use the same state space. In general, the state space of the implementation can be different than that of the specification since the implementations often introduce some components of states that are not used by the specifications. We handle this by relating the states of the concrete implementation with the abstract specification via an abstraction function. The abstraction function is a *total* mapping from Σ_C, the state space of the implementation C, *onto* Σ_A, the state space of the specification A. That is, every state in C is mapped to a state in A, and correspondingly, every state in A is an image of some state in C. All definitions and theorems in this chapter are readily extended with respect to the definition of the abstraction function. The

soundness and completeness of these abstraction functions are discussed in detail in [11].

Henceforth, let C be an implementation and A a specification.

Definition. C is a *refinement* of A, denoted $[C \subseteq A]_{init}$, iff every computation of C that starts from an initial state is a computation of A.

Definition. C is an *everywhere refinement* [5] of A, denoted $[C \subseteq A]$, iff every computation of C is a computation of A.

A fault is a perturbation of the system state. Here, we focus on transient faults that may arbitrarily corrupt the process states. The following definition captures a standard tolerance to transient faults.

Definition. *C is stabilizing to A* iff every computation of C has a suffix that is a suffix of some computation of A that starts at an initial state of A.

This definition of stabilization allows the possibility that A is stabilizing to A, that is, A is self-stabilizing.

We define a wrapper to be a system over Σ and formulate the "addition" of one system to another in terms of the operator $[]$ (pronounced "box") which denotes the union of automata.

Let A and C be distributed systems composed of processes A_i and C_i respectively; i.e., $A = ([]\ i ::\ A_i)$ and $C = ([]\ i ::\ C_i)$. We say that a wrapper W_i for each process A_i is local and atomic iff W_i when executed self-stabilizes (if its state is corrupted) and corrects A_i to a good state (locally consistent state) in a single step.

3 Adopting Ordinary Refinements for Specification-based Design

In this section we prove that we can adopt ordinary refinements for the specification-based design of stabilization under suitable conditions. One of these conditions is the use of a layered composition structure of the processes in the system. Stabilization through composition of layers asserts that the correction and the corruption relations are to the same direction and form a directed acyclic graph. *Corruption relation* denotes for each process in a bad state which other processes it can corrupt. That is the corruption relation constrains the processes an uncorrected process can potentially corrupt. *Correction relation* denotes for each process the prior correction of which other processes its correction depends on. That is, the correction relation constrains the order in which correction must occur. In cases where the correction and corruption relations are in reverse directions, persistent corruption cycles may be formed: even though all the processes are individually stabilizing, the system may fail to stabilize due to the continuous introduction of corruptions to the system via these corruption cycles. However, in the case of layered composition since both correction and corruption relations are in the same direction, corruption cycles do not exist. Therefore, in this case, if all processes are individually stabilizing, then the system as a whole is also stabilizing [10].

Theorem 1 formally states the conditions under which ordinary refinements are usable for the specification-based design of stabilization: Given local and atomic wrappers (premise 3) that achieve stabilization of the abstract system (premise 1), ordinary refinement of the application code at each process (premise 2) when composed with everywhere refinement of the abstract wrapper (premise 4) —provided that the layered composition structure of the abstract is preserved (premise 5)— results into a concrete system that is self-stabilizing to the abstract system specifications.

Theorem 1. If

1. $(\Box i :: [A_i \ \Box \ W_i])$ is stabilizing to A_i,
2. $(\forall i :: [C_i \ \subseteq \ A_i]_{init})$,
3. $(\forall i :: [W_i' \ \subseteq \ W_i])$,
4. $(\forall i :: W_i$ is local and atomic), and
5. the correction & corruption relations of the abstract system are to the same direction, and the concrete system preserves the correction & corruption relations of the abstract

then $(\Box i :: [C_i \ \Box \ W_i])$ is stabilizing to $(\Box i :: A_i)$.

Proof. From premises 3 and 4 it follows that $(\forall i :: W_i'$ is local and atomic), and hence C_i cannot interfere with the recovery strategy of W_i'. Thus, from premises 2, 3, and 4, it follows that $(\forall i :: [C_i \ \Box \ W_i]$ is stabilizing to $[A_i \ \Box \ W_i])$. Even though, at this point we have stabilization at the process level, the system as a whole may fail to be stabilizing due to corruption cycles. Premise 5 takes care of this concern as we discussed above. The conclusion follows from this result and premise 1. □

Theorem 1 shows that an ordinary refinement suffices for the fault-intolerant application, and a self-stabilization preserving refinement suffices for the wrappers. The reason we use a stabilization-preserving refinement (e.g., everywhere refinements) for the wrapper is to ensure that the concrete wrapper is able to stabilize from the corruption of its variables. Since there are a lot of tool support for ordinary refinements, refinement of the tracking algorithm can be done automatically via a compiler. Since the wrappers are small and simple their proof of self-stabilization can be achieved easily even at the implementation level.

4 Refinement of Stalk to the Implementation Level

In this section, we present a refinement of the abstract Stalk program to the TinyOS implementation level by showing that Theorem 1 is applicable for this refinement. We start by recalling some of the properties of Stalk and pointing out which concepts of Theorem 1 they correspond to. We then continue with a discussion of the refinement to the implementation level.

4.1 Brief Summary of Stalk

For achieving scalability, STALK employs a hierarchical structure by using a hierarchical partitioning of the sensor network into clusters based on radius.

Input: $\mathbf{object}_i$
eff: if $c \neq i \ \ \wedge \ \ lvl(i) = 0$ then
$c := i$
$gtime := now + g$

Output: $\mathbf{send\ (gquery)}_{i,j}$
pre: $j \in gnbrquery$
eff: $gnbrquery := gnbrquery - \{j\}$
if $gnbrquery = \emptyset$ then
$gtime := now + g * r^{lvl(i)}$

Input: $\mathbf{receive\ (gquery)}_{j,i}$
eff: if $p = h(i)$ then
$gqack := j$

Output: $\mathbf{send\ (ack\_gquery)}_{i,j}$
pre: $gqack = j$
eff: $gqack := \perp$

Input: $\mathbf{receive\ (ack\_gquery)}_{j,i}$
eff: if $c \neq \perp \ \ \wedge \ \ p = \perp$ then
$p := j$

Output: $\mathbf{send\ (grow)}_{i,j}$
pre: $now = gtime \ \ \wedge \ \ c \neq \perp \ \wedge$
$((j = p \ \ \wedge \ \ p \in nbr(i)) \ \ \vee \ \ (j = h(i) \ \ \wedge \ \ p = \perp))$
eff: if $p = \perp$ then
$p := h(i)$
$gtime := \infty$

Input: $\mathbf{receive\ (grow)}_{j,i}$
eff: $c := j$
if $lvl(i) = MAX$ then
$p := i$
if $p = \perp$ then
$gnbrquery := nbr(i)$

Fig. 1. Stalk protocol: grow actions at process i

Input: **no_object**$_i$
eff: if $lvl(i) = 0 \ \wedge \ c \neq \perp$ then
$c := \perp$
$stime := now + s$

Output: **send (shrink)**$_{i,j}$
pre: $now = stime \ \wedge \ c = \perp \ \wedge \ j = p$
eff: $p := \perp$
$stime := \infty$

Input: **receive (shrink)**$_{j,i}$
eff: if $c = j$ then
$c := \perp$
$stime := now + s * r^{lvl(i)}$

Fig. 2. Stalk protocol: shrink actions at process i

The tracking structure is a path rooted at the highest level of the hierarchy. Each process in the *tracking path* has at most one child, either at its level or one below it in the hierarchy, and the mobile object resides at the leaf of the tracking path, at the lowest level. Each process in the path points to a process that is generally closer to the object and has more recent information about its location.

We implement move-triggered updates to the tracking path by means of two operations, *grow* and *shrink*. The grow operation enables a path to grow from the new location of the object to increasingly higher levels of the hierarchy and connect to the original path at some level. The shrink operation cleans branches deserted by the object. Shrinking also starts at the lowest level and climbs to increasingly higher levels.

A hierarchical partitioning of a network inevitably results in multi-level cluster boundaries: even though two processes are neighbors they might be contained in different clusters at all levels (except the top) of the hierarchy. If a process were to always propagate grows and shrinks to its clusterhead, a small movement of the object back and forth across a multi-level cluster boundary could result in work proportional to the size of the network rather than the distance of the move. To resolve this "dithering" problem, Stalk allows one *lateral link* per level in our tracking path. A process occasionally connects to the original path with a lateral link to a neighboring process rather than by propagating a link to its parent in the hierarchy. Stalk limits the lateral link count per level in order not to upset the locality properties of the find operation.

To implement **Tracker**, each process i maintains a child pointer c, a parent pointer p, a set *gnbrquery* to keep track of which neighbors it has send a query in the last invocation of grow, a variable *gqack* to keep track of which neighbor to reply to, a grow timer *gtime*, and a shrink timer *stime*. In the initial states,

$i.c = i.p = \perp$ and $i.gtime = i.stime = \infty$ for all i. The grow actions are presented in Figure 1 and the shrink actions are presented in Figure 2.

In order to correct for the case where a process i may have a valid child but no parent or a valid parent but no child, Stalk uses two simple actions as in Figure 3. In order to detect and dissociate a child at process i, Stalk employs a heartbeat mechanism as in Figure 4. The actions in Figure 3 are stateless (they do not introduce any new state), and the actions in Figure 4 introduces only one variable (a soft-state variable) to keep track of a timeout.

Internal: **start-shrink**$_i$
pre: $(c = \perp \ \wedge \ p \neq \perp$
$\wedge \ stime \notin [now, now + s * r^{lvl(i)}])$
$\vee [p \in nbr(i) \ \ \wedge \ \ c \in nbr(i)]$
eff: $c := \perp$
$stime := now + s * r^{lvl(i)}$

Internal: **start-grow**$_i$
pre: $c \neq \perp$
$\wedge \ p = \perp \ \ \wedge \ \ gtime \notin [now, now + g * r^{lvl(i)}]$
eff: if $lvl(i) = MAX$ then
$p = i$
if $p = \perp$ then
$gnbrquery := nbr(i)$

Fig. 3. Stalk protocol: correction actions at process i

4.2 Application of Theorem 1 to Stalk

Stalk provides local specifications for the fault-intolerant tracking program: The Tracker$_i$ automata presented in Figures 1 and 2 corresponds to A_i in Theorem 1. Stalk also provides local and atomic wrappers for each Tracker$_i$: The parallel composition of the correction actions in Figures 3 and 4 corresponds to W_i in Theorem 1. Since, in [1], we proved that Tracker$_i$ composed with the correction actions are self-stabilizing, premise 1 is satisfied. Since the correction actions for the Tracker$_i$ automata are all local and atomic (they put Tracker$_i$ in a locally-consistent state in one step), premise 4 is satisfied.

Stalk imposes a static layered structure on the processes: There is no communication from a higher level process to a lower level process; the direction of communication is from lower level processes to higher level processes. Due to this structural constraint, the same layered composition structure is applicable at both the abstract and concrete systems; hence, premise 5 is satisfied.

Next, we consider the refinement of Stalk to the implementation level. In order for Theorem 1 to be applicable, we need to show that premises 2 and 3 are satisfied by our refinement of Stalk.

Output: **send (heartbeat)**$_{i,j}$
pre: $now = next \ \wedge \ j = p$
eff: $next := now + b * r^{lvl(i)}$

Input: **receive (heartbeat)**$_{j,i}$
eff: if $c = \perp$ then $c := j$
if $c = j$ then
$timeout := now + (b + 2\delta m/r) * r^{lvl(i)}$

Internal: **timeout_expire**$_i$
pre: $now = timeout \ \wedge \ c \neq \perp \ \wedge \ c \neq i$
eff: $c := \perp$

Internal: **heartbeat_set**$_i$
pre: $p \neq \perp \ \wedge \ next \notin [now, now + b * r^{lvl(i)}]$
eff: $next := now + b * r^{lvl(i)}$

Internal: **timeout_set**$_i$
pre: $c \neq \perp \ \wedge \ c \neq i$
$\wedge \ timeout \notin [now, now + (b + 2\delta m/r) * r^{lvl(i)}]$
eff: $timeout := now + (b + 2\delta m) * r^{lvl(i)}$

Fig. 4. Stalk protocol: heartbeat actions at process i

Premise 2 asserts that the implementation of the $\mathsf{Tracker}_i$ automata should be a refinement from the initial states. Since there are a lot of tool support for ordinary refinements, refinement of the tracking algorithm can be done automatically via a compiler. For example, the IOA toolkit [12] supports the design, analysis, verification, and refinement of programs written in I/O automata notation. The toolkit includes analysis tools such as the IOA simulator [13] and interfaces to theorem-proving tools [14] as well as compilers for generation of distributed code in commercial programming languages [15]. Even if the implementation of $\mathsf{Tracker}_i$ automata is performed manually, the verification process for ordinary refinements are, in general, easier than that of fault-tolerance preserving and compositional refinements. Since an ordinary refinement from initial states of the $\mathsf{Tracker}_i$ automata is sufficient, one does not have to consider refinements from every state for the purposes of this implementation.

Premise 3 asserts that the abstract wrappers should be everywhere refined. Since sensor nodes [16] have a single thread of control, the concrete level wrappers are made atomic easily by making them to run till completion upon invocation. Hence, if we prove self-stabilization of the concrete wrappers, then this implies that the concrete wrappers are everywhere refinements of the abstract ones. Since the wrappers are small and simple, their proof of self-stabilization can be achieved easily even manually, without any tool support. Model-checking based approaches may also be used for this purpose: For example, [17] can ac-

cept a wrapper written in C language as input, and check the self-stabilization properties of the wrapper.

(Remark:) An interesting research question is the feasibility of automating the translation process from the abstract wrappers to the concrete wrappers. To implement the abstract wrapper at the concrete level, the code translator may use the abstraction function from C to A in the reverse direction. Note that the wrapper synthesized for the abstract model A is readily available for mapping back to C because the abstraction function is defined as onto. (In fact in [18] a similar method is presented for mapping the counterexamples in the abstract model to counterexamples in the concrete model.) Since abstract wrappers are often stateless (as in Figure 3), in these cases the translator would only be responsible for cleaning out the extra implementation state it introduces at the concrete. Soft-state approaches and watch-stop timer based reset approaches might be useful for automating this cleaning task. *(End of remark.)*

Since all the premises are satisfied, we can conclude, by a simple application of Theorem 1, that the resultant implementation of Stalk at the TinyOS level is self-stabilizing to the abstract specifications.

5 Related Work

In this section, we review the previous work on fault-tolerance preserving refinements and compositional frameworks for self-stabilization.

5.1 Fault-tolerance preserving refinements

Our previous work on stabilization preserving refinements. We have shown in [5] that refinements in general are not fault-tolerance preserving, that is, even though A is fault-tolerant, a refinement C of A may not be fault-tolerant. We are therefore led to considering special classes of refinements. In our previous research, we have identified two fault-tolerance preserving refinements: everywhere refinements [5] and convergence refinements [6].

Intuitively speaking, everywhere refinements demand that the implementations always satisfy the specifications from every state. Further, for effective design of fault-tolerance in distributed systems, we identify the subclass of *local everywhere refinements*: these refinements are decomposable into parts each of which must always be satisfied by some system process from *all* of its states without relying on its environment (including other processes).

Intuitively speaking, convergence refinement implies that even in the unreachable states the computations of the concrete system C track the computations of the abstract system A, although some states that appear in the computations of A may disappear in the computations of C, and hence, C preserves convergence properties (e.g., stabilization) of A.

In contrast to previous work on fault-tolerance preserving refinements, we have shown that the refinements we have identified have nice compositionality properties making them suitable for specification-based design of fault-tolerance.

For example, convergence refinement enables a non-stabilizing implementation C to be made stabilizing without knowing the implementation details of C but knowing only an abstract specification A that C satisfies. More specifically, given C that is a convergence refinement of A, first stabilization of A is designed by devising an abstract wrapper W for A. Stabilization of C is then achieved by adding to C any convergence refinement of W; the refined wrapper is oblivious to the implementation details of C.

The lightweight method we present in this paper enables the use of ordinary refinements (for which a lot of compiler/tool support exists) in the specification-based design of stabilization in lie of an everywhere or convergence refinement.

Method by Z. Liu and M. Joseph. Liu and Joseph [7] have considered designing fault-tolerance via transformations. In their work, an abstract program A is refined to a more concrete implementation C and then based on the refined program C and the fault actions F that are introduced in the refinement process, further precautions (such as using a checkpointing & recovery protocol) are taken to render C fault-tolerant. Liu and Joseph design the tolerance based on the concrete program, while we design our wrappers based on the abstract program.

Method by L. Lamport and S. Merz. In [8], Lamport and Merz claim that there is no need for a special technique for formal specification and verification of fault-tolerance systems, and that refinement of fault-tolerance programs could be achieved using temporal logic of actions (TLA) and a hierarchical proof method.

Towards this end, they show how a message-passing Byzantine agreement program (of [19]) can be derived from its high-level specification. (The authors, however, do not discuss how their example can be generalized into a method for designing arbitrary fault-tolerant programs.) They first present three specifications for the Byzantine agreement program: a high-level problem specification, a mid-level specification of the algorithm, and a low-level specification for message-passing model. Then they prove that each specification implements the next-higher one.

The authors claim that little ingenuity is required for proofs of refinements since a hierarchical proof strategy is adopted. However, it should be noted that a considerable amount of ingenuity is still required for coming up with the refinement programs in the first place. The authors also admit in the discussion section of the paper that their method is "not yet feasible for reasoning at the level of executable code, except in special applications or for small parts of a system."

Fault-tolerance preserving atomicity refinements. Fault-tolerance preserving refinements have been studied in the context of atomicity refinement in [20, 21], whereas in our work we study them in the more general context of computation-model refinement.

McGuire and Gouda [22] have developed an execution model that can be used in translating abstract network protocol specifications written in a guarded-command language into C programs using Unix sockets. Their framework cannot

handle arbitrary state corruptions we considered here, and only allows the following faults: message loss, message ordering, and message duplication.

Semantics of fault-tolerance preserving refinements. Leal [23] has also observed that refinement tools are inadequate for preserving fault-tolerance. The focus of his work is on defining the semantics of tolerance preserving refinements of components.

5.2 Compositional frameworks for self-stabilization

Scalable design of stabilization through composition idea has been around for a long time [10]. In the traditional stabilization by layers approach lower-level components are oblivious to the existence of higher-level components, and higher-level components can read (but not write) the state of a lower-level component. Components can corrupt each other, but only in a predetermined controlled way since lower-level components cannot be affected by the state of higher-level ones. Also, the order in which correction must take place is the same direction, the correction of higher-levels depend on that of lower-levels. Adaptive programming [24] is similar to stabilization by layers, except that the components depend on an environment that may change. If the environment achieves a fixed point for a sufficiently long time, the components stabilize with respect to it.

In [25], a compositional framework for constructing self-stabilizing systems is proposed. The framework explicitly identifies for each component which other components it can corrupt (corruption relation). Additionally, the correction of one component often depends on the prior correction of one or more other components, constraining the order in which correction can take place (correction relation). A global reset [26] is potentially avoided and fault-containment is enabled when possible by using the correction and corruption relations to check and block certain components to prevent formation of fault-contamination cycles.

Depending on what is actually known about the corruption and correction relations, the framework offers several ways to coordinate system correction. In cases where the correction and corruption relations are in reverse directions, persistent corruption cycles may be formed: even though all the components are individually stabilizing, the system may fail to stabilize due to the continuous introduction of corruptions to the system via these corruption cycles. By employing blocking coordinators, the framework breaks these malicious cycles. In cases where both correction and corruption relations are in the same direction, no cycle forms and there is no need for blocking. By including both correction and corruption relations, the framework in [25] subsumes and extends other compositional approaches, such as layered composition, where correction and corruption relations are to the same direction.

In our paper, while developing lightweight and local refinements for designing specification-based self-stabilization to tracking, we restricted our work to the systems where both the correction and corruption relations are to the same direction in both the abstract and the concrete levels. This way we did not have to deal with addition of extra coordinators and blocking at the concrete system level. By adopting the framework in [25] for our refinement method, we can relax our

layered composition assumption and allow arbitrary compositions of processes. Using the knowledge of correction-corruption relations between the processes, we can instantiate a corresponding coordinator to ensure that stabilization of processes compose at the abstract level. In fact, then these coordinators become part of the wrappers at the abstract, and by refinement of these wrappers we can achieve stabilization at the concrete level.

6 Concluding Remarks

In this paper we showed that we can use ordinary refinements (for which a lot of compiler/tool support exists) and still achieve a specification-based design of stabilization under suitable conditions. To this end, we assumed that (1) wrappers are local to each process and are atomic, and (2) the concrete system preserves the layered composition structure of the abstract system. Using these two conditions, we showed that an ordinary refinement suffices for the fault-intolerant application, and a self-stabilization preserving refinement suffices for the wrappers. Another advantage of our specification-based design is that it enables a posteriori addition of stabilization. That is, starting with a concrete implementation C, it is possible to add fault-tolerance to C by first designing an abstract tolerance wrapper W using solely an abstract specification A of C, and then adding a concrete refinement W' of W to C.

We have illustrated this lightweight method for specification-based design on the Stalk protocol for wireless sensor networks. We believe that for a rich class of wireless sensor network applications the two conditions we have identified for the applicability of our method holds naturally. In future work, we will provide an actual demonstration of Stalk on the motes as part of our NSF-funded ongoing project on pursuer-evader tracking in wireless sensor networks. Also, we will investigate other wireless sensor network applications where our method is applicable for achieving stabilization at the implementation level.

References

1. Demirbas, M., Arora, A., Nolte, T., Lynch, N.: A hierarchy-based fault-local stabilizing algorithm for tracking in sensor networks. 8th International Conference on Principles of Distributed Systems (OPODIS) (2004) 299–315
2. : Crossbow technology, Mica2 platform. `www.xbow.com/Products/Wireless_Sensor_Networks.htm`
3. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesc language: A holistic approach to networked embedded systems. In: PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation. (2003) 1–11
4. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for network sensors. ASPLOS (2000) 93–104
5. Arora, A., Demirbas, M., Kulkarni, S.S.: Graybox stabilization. Proceedings of the International Conference on Dependable Systems and Networks (ICDSN) (July 2001) 389–398

6. Demirbas, M., Arora, A.: Convergence refinement. Proceedings of the International Conference on Distributed Computing Systems (ICDCS) (July 2002) 589–597 Best paper(1st/335).
7. Liu, Z., Joseph, M.: Transformations of programs for fault-tolerance. Formal Aspects of Computing **4**(5) (1992) 442–469
8. Lamport, L., Merz, S.: Specifying and verifying fault-tolerant systems. Third Symposium on Formal Techniques in Real Time and Fault Tolerant Systems, LNCS 863 (1994) 41–76
9. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Springer-Verlag (1990)
10. Dolev, S.: Self-Stabilization. MIT Press (2000)
11. Demirbas, M.: Scalable design of fault-tolerance for wireless sensor networks. PhD thesis, The Ohio State University (2004)
12. Garland, S.J., Lynch, N.A.: Using i/o automata for developing distributed systems. Foundations of Component-Based Systems (2000) 285–312
13. Kaynar, D.K., Chefter, A., Dean, L., Garland, S., Lynch, N., Win, T.N., Ramirez, A.: The ioa simulator. Technical Report 843, MIT Laboratory for Computer Science (2002)
14. Garland, S., Guttag, J.V., Horning, J.: An overview of larch. Functional Programming, Concurrency, Simulation and Automated Reasoning (1993)
15. Tauber, J.A.: Verifiable Code Generation from Abstract I/O Automata. PhD thesis, MIT (2003)
16. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for network sensors. ASPLOS (2000) 93–104
17. Hatcliff, J., Dwyer, M.B., Pasareanu, C.S., Robby: Foundations of the bandera abstraction tools. (2002) 172–203
18. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Computer Aided Verification. (2000) 154–169
19. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programming Languages and Systems (1982)
20. Nesterenko, M., Arora, A.: Stabilization-preserving atomicity refinement. 13th International Symposium on Distributed Computing (DISC) (1999)
21. Beauquier, J., Datta, A.K., Gradinariu, M., Magniette, F.: Self-stabilizing local mutual exclusion and daemon refinement. International Symposium on Distributed Computing (2000) 223–237
22. McGuire, T.M.: Correct implementation of network protocols. PhD thesis, University at Texas at Austin (2004)
23. Leal, W.: A Foundation for Fault Tolerant Components. PhD thesis, The Ohio State University (2001)
24. Gouda, M.G., Herman, T.: Adaptive programming. IEEE Transactions on Software Engineering **17** (1991) 911–921
25. Leal, W., Arora, A.: Scalable self-stabilization via composition and refinement. In: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS), IEEE (2004)
26. Arora, A., Gouda, M.G.: Distributed reset. IEEE Transactions on Computers **43**(9) (1994) 1026–1038