Software Composition and Verification for Sensor Networks

P. Völgyesi [a], M. Maróti [b], S. Dóra [b], E. Osses [b] and Á. Lédeczi [b]

[a] *Embedded Information Technology Research Group, Hungarian Academy of Sciences – Budapest University of Technology and Economics, Magyar tudósok körútja 2, Budapest, 1117, Hungary*

[b] *Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN 37221, USA*

Abstract

Component-based design has become a necessity for networked embedded systems where hardware platforms come in a great variety and evolve extremely rapidly. Operating system components and higher level middleware services call for modular software construction along clear interfaces. The way we describe these interfaces and process the captured information is of crucial importance to exploit the benefits of component-based design. In this paper we present a model based approach to the development of embedded applications with a special emphasis on interface specification. The proposed formalism captures the temporal and type aspects of interfaces and supports composition and verification of components. Along with the formal definition of the proposed interface language and component compatibility rules, we present a modeling environment targeting TinyOS, a representative embedded operating system. Two prototype tools are also described that check the composability of components based on their interface models and verify that the implementation of a component matches its formal model, respectively.

1 INTRODUCTION

Component-based design is increasingly viewed as the cornerstone of software engineering. The advantages of using components stem from the fact that they can be developed and tested in isolation, and systems can be built and updated incrementally. When they are designed with adequate generality, components can be reused in different applications. Component-based design has become especially important for networked embedded systems where hardware platforms and operating systems are characterized by a rapid pace of innovation.

Preprint submitted to Elsevier Science *27 August 2004*

This is best exemplified by the advent of TinyOS [11], an operating system specifically designed for sensor networks, and nesC [14], its programming language. Even the most basic system modules of TinyOS are components that can be augmented or replaced for different applications and/or platforms. The nesC language defines a component model that relies on bidirectional interfaces and admits an efficient implementation that avoids dynamic component creation. TinyOS applications are statically linked graphs of event-driven components. Typically, the same application image is executed on all (or most of) the nodes of the network. Full-blown sensor network systems are built from hundreds of intricately interacting components through thousands of component interfaces. Manual wiring of components, a tedious and error prone task in nesC, can be automated by composition tools, such as Gratis [13]. However, the truly challenging and especially missing ingredient for the development of mission critical, large scale sensor network applications is component and composition verification.

Verification of embedded systems has an extensive research literature covering formal verification and model checking methods [6]. Nevertheless, only a selected few approaches address the special needs of sensor networks, such as the theory of Input Output Automata [16] and that of Interface Automata [1]. The automata-theoretic approach lends itself naturally to the study of networked sensor applications, because of their inherent event-driven nature. Existing methodologies do not exploit the massively componentized and hierarchical structure of nesC programs. In such designs the reactions between moderately sized software components are restricted by the single flow of control within the application as opposed to a distributed system with asynchronously scheduled processes [4]. This work is focused on the modeling and light weight verification of such component systems and does not claim unrestricted applicability in other domains.

Most programming errors during application composition are either the result of incorrectly used components or the bad interaction of multiple components, some of which could be operating system components not even considered by the developer. We address these sources of programming errors by introducing a hierarchical component verification formalism based on the Interface Automata language and by extending the Gratis environment with a prototype verification tool for TinyOS applications.

In the following sections we overview TinyOS and Gratis. Then we formally define hierarchical interface automata, our formalism for modeling component interfaces. Next we define the composition and compatibility verification of hierarchical interface automata. We describe how these automata can be used to validate existing hand-written components, as well as assemblies of components. Finally, we illustrate the use of the proposed formalism in the extended Gratis environment.

2 TINYOS

TinyOS is a component-based configurable operating system with a very small footprint specifically designed for severely resource constrained devices such as the nodes in a typical sensor network [11]. TinyOS is a large set of software components implementing the basic functionalities that an application might need from the given device, such as basic I/O, timers, wireless communication etc. Components can contain other components in a hierarchical fashion. Each application consists of application-specific components written by the application designer and a subset of the TinyOS components. This way an application-specific TinyOS instance is created for each application providing only the services the application needs, which conserves precious system resources.

A TinyOS application consisting of a set of interconnected components is scheduled by a simple FIFO-based non-preemtive scheduler. Components communicate with each other through commands and events. Commands propagate downward; they are issued by higher level components to lower level ones. Events propagate upward; they are signaled by lower level components and handled by higher level ones. Events at the lowest level are generated by the hardware itself in the form of interrupts.

Commands are typically handled by updating the state of component, possibly posting a task for later execution and possibly issuing commands to lower level components. An event handler can also modify the state of the component, signal higher level events or call lower level commands. Notice that commands cannot signal events to avoid cycles. Tasks are the worker bees of TinyOS. They can issue commands, signal events and post other tasks. Tasks are intended to do a short amount of processing and return. They can only be preempted by events not by other tasks. This task model enables TinyOS to have a single call stack.

The latest version of TinyOS (version 1.1) is implemented in nesC [14]. nesC, an extension of C, is a new language developed specifically to support the TinyOS model of computation. It disallows dynamic memory allocation and dynamic dispatch making nesC programs statically analyzable and optimizable.

The three major building blocks of a nesC application are interfaces, modules and configurations. An interface is a set of related events and/or commands. In other words, an interface is a set of function declarations. The provider of an interface needs to implement the commands, while the user of the interface needs to implement the events.

Modules and configurations are both components. Modules are the elementary building blocks; they have actual procedural nesC code associated with them specifying their functionality. Configurations are the composite components; they contain modules and/or other configurations and the wiring specification connecting the various interfaces of the contained components together. Every nesC (and TinyOS) application has a single top level configuration [14].

3 GRATIS

A TinyOS application is a hierarchical component assembly where component configurations, i.e. wiring specifications, interface declarations and module implementations are specified in numerous text files. Graphical representation of the same information increases the readability and understandability of the application architecture and helps in avoiding configuration errors, such as the omission of the wiring specification of one or more interfaces of a component.

The Graphical Development Environment for TinyOS (Gratis) is a typical application of Model Integrated Computing (MIC) in general, and the Generic Modeling Environment (GME), in particular [12]. GME is a metaprogrammable toolkit for creating domain-specific modeling environments. GME metamodels specify the modeling language of the application domain. They are used to automatically configure GME for the domain, that is, to create a modeling environment that has native support of the target modeling language.

GME models take the form of graphical, multi-aspect, attributed entity-relationship diagrams. Their syntax is defined by the metamodels specified in a UML class diagram-based notation. The static semantics of a model are specified by OCL constraints [17] that are also part of the metamodels. They are enforced by a built-in constraint manager during model building time. The dynamic semantics are applied by the model translators, i.e. by the process of translating the models to source code, configuration files, database schema or any other artifact the given application domain calls for.

This approach fits component-based software development very nicely. The interface of the individual components can be modeled along with a link to their implementation. The model editor can enforce the composition rules to make sure that only valid component assemblies are allowed. More sophisticated analysis can be performed by interfacing to outside tools. Finally, model translators can generate the glue code that ties the final system together.

The metamodel of Gratis defines the mapping of TinyOS concepts to GME concepts, as shown in Figure 1. The three basic building blocks of Gratis models are *interfaces*, *modules* and *configurations*. An interface consists of a

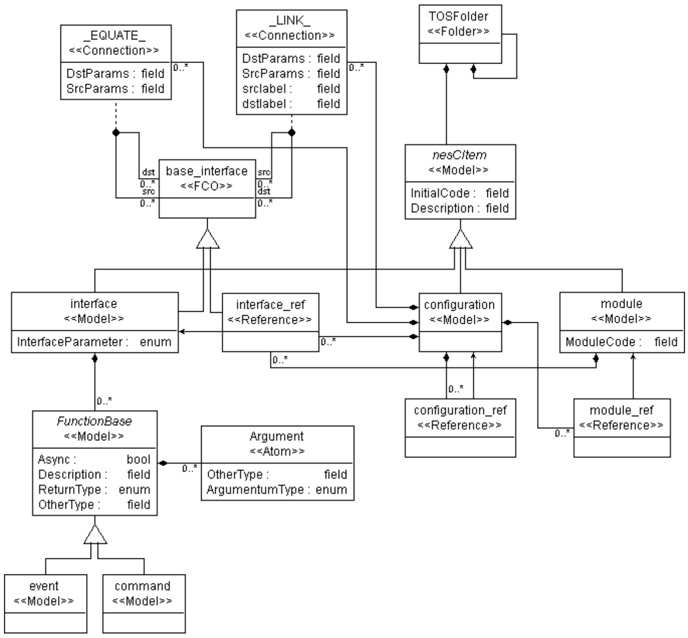

Fig. 1. The simplified metamodel of Gratis.

set *events* and *commands*. Both events and commands are functions. The return type is captured by a textual attribute, while the *arguments* are modeled with contained objects each having its own type declaration. A module contains a set of interface references (*interface_ref*) and its nesC code as a textual attribute. A reference is a graphical object that points to another object contained elsewhere in the model hierarchy. This is captured in the metamodel by a directed connection pointing from the interface to the interface_ref metamodel in Figure 1. Interfaces are declared at the global level and modules do not contain them directly; they just refer to their declaration through the use of references. This allows multiple modules using and/or implementing the same interface declarations. Also, when an interface needs to be modified, it is done at one place and all interface references in all components will refer to the updated interface automatically.

Similarly, configurations contain references to interfaces, modules (*module_ref*) and other configurations (*configuration_ref*). Interface references contained in modules and configurations appear as ports in higher level configurations. Component wiring specifications are expressed in Gratis as connections between interfaces and/or interface ports in configurations. In fact, two different kinds of connections are used in configurations. A *LINK* specifies that a component uses an interface that another provides. An *EQUATE* connection specifies that the interface the given configuration uses/provides is delegated down to a contained component that either implements it or delegates it further down the component hierarchy.

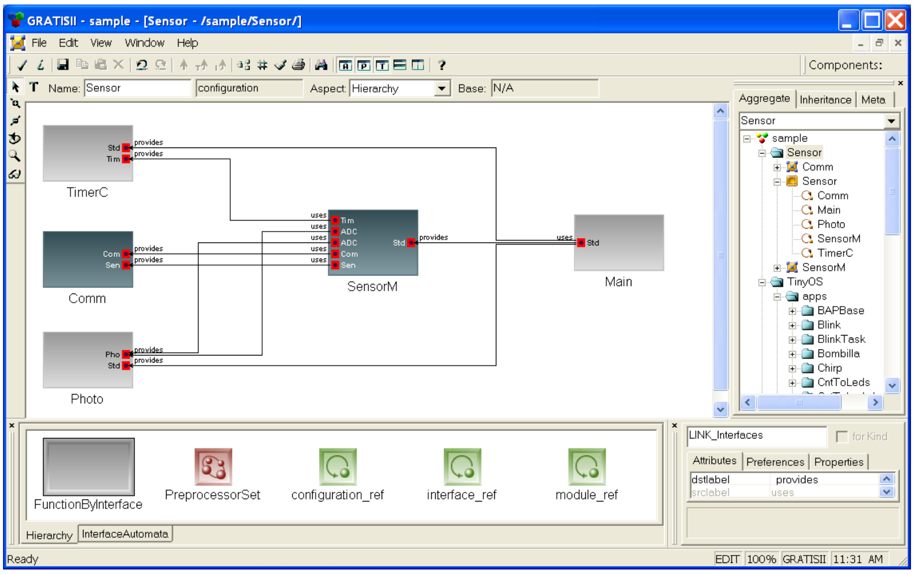

Fig. 2. Top level model of the *Sensor* application in Gratis.

Figure 2 depicts an example application modeled in Gratis illustrating these concepts. This application periodically samples a photo sensor and sends the measured readings to a base station. The participating components are Main, the standard entry point of all applications in TinyOS, TimerC, the abstraction layer of hardware timers, Photo, the photo sensor driver, Comm, the standard wireless communication protocol stack of TinyOS, and SensorM, the sole application specific module. Module and configuration components are depicited in dark and light colors, respectively. The ports of the components are provided and used interfaces. This example will be used throughout the paper.

The only information captured textually in Gratis is the procedural code of module implementations. The model translator generates all the nesC files containing interface, module and configuration specifications automatically. Keeping the graphical models and the corresponding nesC files in synch is a challenge, especially because a large code base of TinyOS components exists in text form only. Therefore, the Gratis model translator is bi-directional; not only does it generate the nesC files from graphical models, but it is also capable of parsing existing source files and building the corresponding models automatically. The main use of this parsing feature is to automatically generate the graphical equivalent of the TinyOS system components and to provide them as a library to the user in the Gratis environment. This library can then be refreshed when new TinyOS versions become available without any modifications to existing graphical application models.

4 TEMPORAL MODELS OF COMPONENT INTERFACES

Traditional programming languages and interface description methods—such as CORBA IDL—capture only the type aspects of software components. The access points of a given component are enumerated along with their accepted and returned parameter types in terms of values and domains. TinyOS and its implementation language *nesC* [14] is no exception to this: component interfaces are defined by a set of function declarations. Compatibility checking provided by compilers guarantees that the user of a function provides the required parameters and handles the returned value in a type-safe manner.

Even in trivial applications, the access points of a software component are not isolated, dependencies and complex relationships might impose additional constraints on the use of their services. Typical patterns—such as *initialization before use* —can be found in almost every component. A component providing communication services may have more restrictions that are inherent in the communication protocol. Even if the legal order and dependencies of the function calls are described in the documentation of the component as informal rules, automatic tools and formal methods cannot be developed to verify these constraints.

Graphical models of traditional interfaces enable us to understand and build complex applications, however they do not extend the information captured in the textual representation. Effective composition and reuse of software components demands deeper understanding and specification of component interfaces. Our proposed formalism—based on the Interface Automata language [1]—captures the dynamic aspects of component interfaces and enables us to describe more complex behavior.

In the following sections, formal rules of interface compatibility will be given along with the description of practical methods for verification and validation of component interfaces and interaction. We have also implemented a proof-of-concept prototype environment of our interface language that targets the TinyOS platform and augments the previously described Gratis tool [13].

4.1 Interface Automata

Our interface modeling language is based on the definition of Interface Automaton, which we will reproduce here.

Definition 1 *An* interface automaton P *consists of the following elements:*

- $states(P)$*, a set of* states,

- *$inits(P)$, a nonempty subset of $states(P)$, known as the* initial states*,*
- *$ins(P)$, $outs(P)$ and $internals(P)$, mutually disjoint sets of* input*,* output *and* internal *actions. We denote the set of all actions by $acts(P) = ins(P) \cup outs(P) \cup internals(P)$, and*
- *$steps(P)$, a set of steps, where $steps(P) \subseteq states(P) \times acts(P) \times states(P)$.*

A simple example of an interface automaton is given in Figure 3. The model describes the interface of the *Comm* component, which provides communication services to its clients. The component accepts the *init* and the *sendMsg* input actions and signals the *sendDone* output action. However, these actions are not accepted or generated arbitrarily. The legal orders are defined with the help of states $s0$, $s1$ and $s2$, where $s0$ is the initial state.

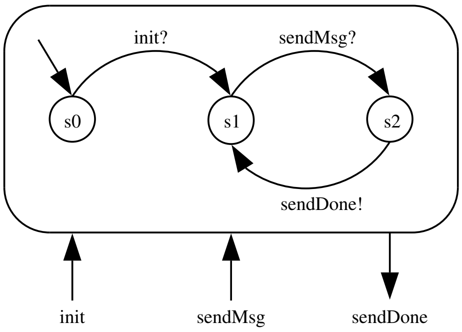

Fig. 3. The Interface Automaton *Comm*.

Proper clients of this component should reckon with the temporal dependencies between the input and output actions of *Comm*. For instance they must not send a second consecutive *sendMsg* action to *Comm* without waiting for a *sendDone* message before. Detailed informal description of compatibility of components is given in Section 4.1 along with the exact definition in Def 6. A trivial compatible component *App* can be constructed by replicating the states and steps of component *Comm*, but inverting the direction of its actions , thus input actions in *Comm* become output actions in *App* and vice versa. In their composition, the two automata are advancing together step by step, always transmitting or receiving an action that is accepted or sent by the other. Formal definition of composite automata will be given in 4.4.

In general, the cross-product of the set of states in the original components generates the state space of the composite automaton. A step in the product might be either a *shared step*, which advances both of the original automata together, as we have seen in the previous paragraph, or an independent step in one of the components. Actions along shared steps become internal actions in the product automaton.

Compatibility analysis must focus on composite states, where one of the original automata initiates a shared step, but the other component is not prepared for accepting this action in its respective state. We denote these composite states as *illegal states*. There are two basic approaches to classify illegal states with respect to compatibility. The "pessimistic" approach defines two components incompatible, if any illegal state is reachable in their composition, i.e. there exists a sequence of steps whose first state is one of the initial composite states and whose last state is the illegal state. The "optimistic" approach considers two components compatible if there is some environment—a third automaton—under which the composite automaton behaves correctly.

The pessimistic approach demands strict compatibility, and it guarantees that independently of additional components, the inspected modules will work together correctly. When the product system is closed—i.e. each action is internal—the pessimistic and optimistic approaches coincide [1]. In this paper we are following the pessimistic approach; a detailed discussion of the optimistic method can be found in [3].

4.2 Hierarchical Interface Automata

Interface Automata have similar limitations to Finite State Machines: in their flat form both languages have scalability problems when describing complex behavior and state space. We propose additional constructs to the original automata language to overcome these problems.

In embedded applications external events from the physical environment might arrive in any moment regardless of the current state of the application. These external events are propagated through the software components via interrupts and function calls. Therefore, to build compatible components, their interface models need to handle these events in all states resulting in a potentionally large number of steps. Hierarchical states enable us to simplify these often incomprehensible models.

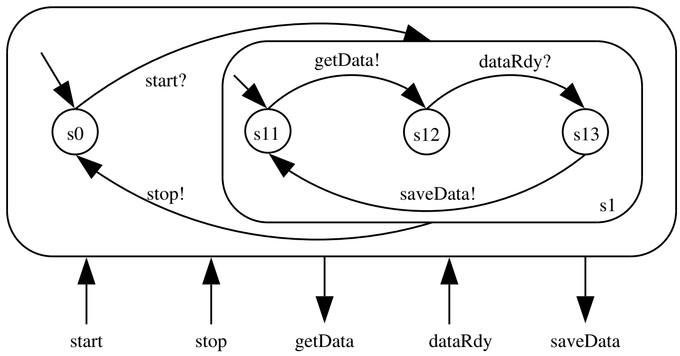

Fig. 4. Hierarchical interface model of *Recorder*.

Figure 4 shows the hierarchical interface model of a simple data logger component, *Recorder*. By receiving a *start* input signal, the component enters into a loop of data acquisition (*getData*, *dataRdy*) using for example a sensor peripheral and data storage (*saveData*). The module must be prepared to receive a *stop* signal and leave the loop at any moment.

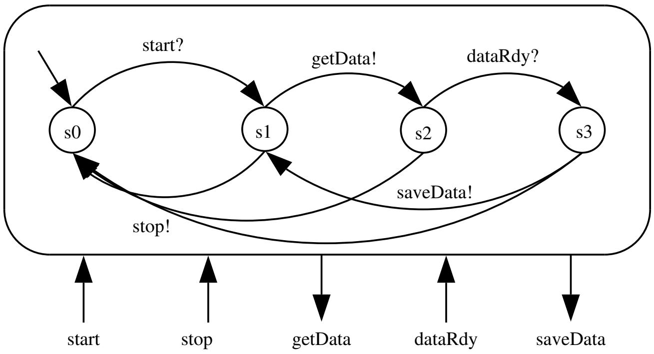

Fig. 5. Flat interface model of *Recorder*.

The flat interface model of the same component is given in Figure 5. Even for this simple component, the benefits of the hierarchical model are noticeable, not only because we have spared two "arrows" in the model, but because it captures the essence of the *stop* signal and preserves visual clarity. If the logic in the recording loop needs refinement, the hierarchy ensures that the stop signal will be handled in the new states, as well, which is not true in the flat scenario.

The original Interface Automata language is a superb formalism for specifying interfaces in event driven systems where each component has its own thread of control and the components engage one another asynchronously via events. However, the concepts of interface automata cannot be mapped to typical embedded applications easily, where the software components are linked together and communicate via function calls. Since our primary goal was to provide interface models for TinyOS components, we had to address the following issues:

- What constitutes an action in these applications? Is it only the function call that conveys information, or its return, as well?
- Interface Automata make decisions based purely on the received actions, unlike in procedural systems where actions are accompanied by quantitative parameters that have strong influence on the control flow.
- In monolithic embedded applications the assumptions of parallel execution and asynchronous message passing no longer hold. Implicit constraints restrict the execution of an automaton that are inherent in sequential execution.

To address the first problem, we have chosen the function calls as the sole representation of actions in the interface model. Our decision was influenced by a feature provided by the *nesC* compiler, which allows to "fan out" a function call to multiple components, that itself raises the question of what the return value of a function means. The second question may be resolved by introducing multiple actions for a given function, however, an extension of the interface language with action parameters is an area of further study.

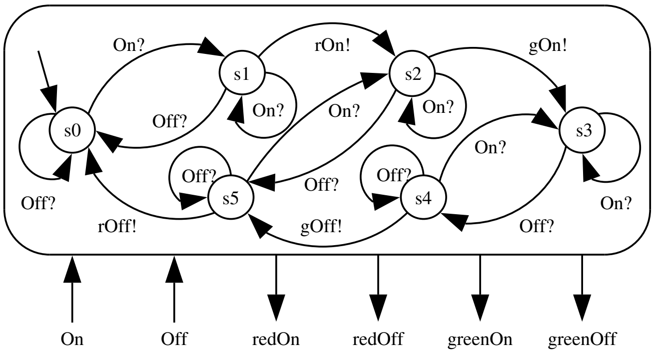

Fig. 6. Interface model of component *LEDs* using the original automata language.

The last problem has driven us to introduce another extension to the original language. Figure 6 depicts the interface automata of a common embedded component that provides access to the display LEDs of the hardware. Our driver is fairly simple: it allows to turn off or on both of its supervised LEDs (red and green). Using the original interface automata language and the pessimistic compatibility rules, the interface becomes incomprehensible, since it needs to handle incoming requests disregarding its current state. In practice, this component would turn on or off all of the LEDs by using a simple sequence of commands in an atomic way. Therefore, we have introduced *non-preemptable states*, which enable us to specify atomic action sequences as shown in Figure 7. Upon entering a non-preemptable state—designated by solid circles—multiple output actions are allowed to be sent before entering a regular state again. Non-preemptable states can be implemented in several ways: the most trivial approach is interrupt masking or the use of mutexes. The *nesC* language and the TinyOS concurrency model has similar constructs, which make a distinction between asynchronous and synchronous code.

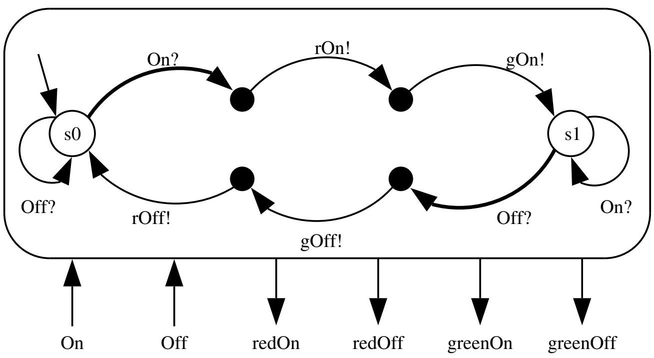

Fig. 7. Interface model of component *LEDs* with non-preemptable states.

The hierarchical representation of interfaces can be transformed automatically to the traditional flat notation, thus the original rules of compatibility are applicable to hierarchical models. The introduction of non-preemptable states, however, needs a slight modification to these formulae. The formal definition of compatibility in Hierarchical Interface Automata is given in the next section. The definition of hierarchical interface automata with non-preemptable states follows.

Definition 2 *A* hierarchical interface automaton P *consists of the following elements:*

- *Elements of regular interface automata as defined in Definition 1.*
- $hstates(P)$*, a set of* hierarchical states*, each of which is a subset of* $states(P)$ *and* $hstates(P)$*. Steps originating from hstates are implicitly defined for each contained sub-state. Steps entering into hstates are implicitly defined for the contained initial sub-state.*
- $npstates(P)$*, a set of non-preemptable states,* $npstates(P) \subseteq states(P)$*. The automaton does not accept input actions in non-preemptable states.*

Adding hierarchy to traditional finite automata is a well known and widely used extension in the domain of reactive systems. The most renowned formalism is Statecharts, presented by Harel [2]. He introduced concurrency and communication along with hierarchy to be able to handle more complex systems. Extended Hierarchical Automata [7], which can be regarded as a kind of abstract syntax of Statecharts address similar weaknesses of traditional FSMs. An Extended Hierarchical Automaton is composed of a set of sequential automata, whose states can be mapped to a set of automata which refine it. A different approach results in a similar structure in [8], where Hierarchical Automata are constructed from elementary automata using the operations $restricted - product$ (communicating parallel composition), $free - product$ (parallel composition with no communication) and $restricted - sum$ (sequential composition). Components are hierarchically structured into super-components in [10], where input and output behavior are separately modeled with the help of finite state machines while internal control flow is described in UML sequence charts. This is in contrast to our notation, where input, output and internal transitions are blended in one automaton.

Hierarchical Interface Automata, as introduced in this section, also employ hierarchy to cope with complexity. Non-preemptable states are the other extension of our formalism, which reflect a unique property—single thread of control—of our target domain. Another distinct, though theoretical feature is that a HIA, like the original interface automaton, never tries to capture the entire state space of the components. Its use is restricted to describe the temporal dependencies among requests and responses.

The operational semantics of a HIA can be defined by a labeled transition system (LTS) [9] after hierarchy is eliminated with an automatic transformation. The transformation builds a flat model by collecting all leaf states and inserting additional transitions based on Definition 2.

4.3 Visual Language Specification

Temporal interface models complement the traditional interfaces of TinyOS components, thus integrating these automaton concepts into the existing Gratis language is the natural choice. The metamodel of the hierarchical interface automata language is shown in Figure 8. The *module*, *interface* and *FunctionBase* meta objects are joint concepts in the original Gratis and the automata language; they connect the type based and temporal models. The multi-aspect capability [12] of the modeling environment enables the separate visualization of the alternative representations of component interfaces. The apparent symmetry in the metamodel stems from the fact that the language can capture the temporal models of both components and interfaces.

TinyOS interfaces and modules can contain states (*mState* and *iState*) that can be nested, and actions (*actionRef* and *FunctionBase*, that can be used to connect states. Special boolean attributes of the state object may designate it as an initial (*IsInit*) or non-preemptable (*IsPreemptable*) state.

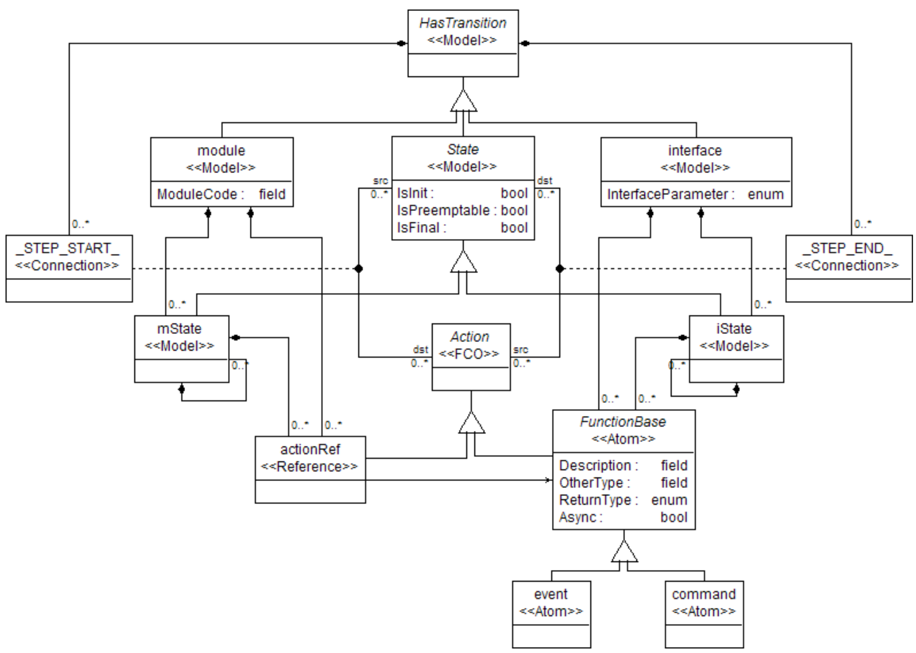

Fig. 8. The metamodel of the Hierarchical Interface Automata language.

We consider two hierarchical interface automata composable if there is no conflict between their actions, thus they only possess common actions which move the product automaton along shared steps.

Definition 3 *Two interface automata P and Q are* composable *if*

$$
\begin{aligned}
&states(P) \cap states(Q) = \emptyset,\\
&internals(P) \cap acts(Q) = \emptyset,\\
&internals(Q) \cap acts(P) = \emptyset,\\
&outs(P) \cap outs(Q) = \emptyset, \text{ and}\\
&ins(P) \cap ins(Q) = \emptyset.
\end{aligned}
$$

Considering Definition 3 one might notice that we made a very important assumption on the state space of the primary automata. Our rules for composition prohibit state interference (i.e. composition of components with shared variables or channels). Although states are defined as abstract artifacts in Definition 2, this assumption is inherent and justified in the domain of TinyOS applications, where shared data areas among components are not allowed [11]. This limitation—enforced by the nesC compiler [14]—is a trade-off for supporting isolated development and testing of TinyOS components.

As we have seen previously, the composite automaton consists of the product of the original (leaf) states and the conjunction of actions reduced by the set of joint actions. Composite steps are defined as for normal interface automata; the only difference is in the special treatment of intermediate states: input actions in the original automata are not accepted while the other component resides in a non-preemptable state.

Definition 4 *If two interface automata P and Q are composable, their* product *$P \times Q$ is defined by*

$$
\begin{aligned}
&states(P \times Q) = states(P) \times states(Q),\\
&npstates(P \times Q) =\\
&\quad \{\, (s,t) \in states(P \times Q) \mid s \in npstates(P) \vee t \in npstates(Q) \,\},\\
&inits(P \times Q) = inits(P) \times inits(Q),\\
&ins(P \times Q) = ins(P) \cup ins(Q) \setminus S,\\
&outs(P \times Q) = outs(P) \cup outs(Q) \setminus S,\\
&internals(P \times Q) = internals(P) \cup internals(Q) \cup S,
\end{aligned}
$$

$$\begin{aligned}
steps(P \times Q) = \Big\{ &((s,t),a,(s',t)) \mid \\
&\Big((s,a,s') \in steps(P) \wedge a \notin S \wedge t \in (states(Q) \setminus npstates(Q))\Big) \\
&\vee\Big((t,a,t') \in steps(Q) \wedge a \notin S \wedge s \in (states(P) \setminus npstates(P))\Big) \\
&\vee\Big((s,a,s') \in steps(P) \wedge (t,a,t') \in steps(Q) \wedge a \in S\Big)\Big\},
\end{aligned}$$

where $S = acts(P) \cap acts(Q)$.

Based on the informal description of illegal states in 4.1, their precise definition is the key step towards understanding compatibility between components. Compared to the informal description of illegal states given earlier, we had to alter the definition because non-preemptable states are excluded from the analysis.

Definition 5 *The set of* illegal states *in the product of hierarchical interface automata* P *and* Q *is defined by*

$$\begin{aligned}
illegals(P \times Q) = \Big\{ &(s,t) \mid (s,t) \in states(P \times Q) \wedge \exists a \in S \\
&\Big(\Big(a \in outs(P) \wedge t \notin npstates(Q) \\
&\quad \wedge \exists (s,a,s') \in steps(P) \wedge \nexists (t,a,t') \in steps(Q)\Big) \\
&\vee\Big(a \in outs(Q) \wedge s \notin npstates(P) \\
&\quad \wedge \exists (t,a,t') \in steps(Q) \wedge \nexists (s,a,s') \in steps(P)\Big)\Big)\Big\}.
\end{aligned}$$

According to the pessimistic approach one has to traverse the composite interface automaton from its initial state to decide on compatibility between the components.

Definition 6 *We consider two hierarchical interface automata,* P *and* Q*,* compatible *if no illegal states in the product automata can be reached from the initial states.*

5 COMPATIBILITY CHECKING

Based on the previous definitions several algorithms can be developed using graph- or game theoretical [1] foundations. Collecting reachable states and analyzing each of them suggests graph traversing logic, while the interaction of two interface automata might be easily solved by a game between the components.

Similarly to the extended interface language, the rules of compatibility have been evolved gradually, thus implementing an re-implementing custom algorithms for these changing ruleset would have burdened our work. We wanted to create a rapid prototyping framework by reducing the time and effort needed to map formal rules into executable algorithms. Note that similar motivations lead us to the development of the Generic Modeling Environment. By analyzing the nature of our formulae, we concluded that logic programming languages provide ideal ground to implement and execute these formal rules and statements.

Based on the visual interface specifications created in the modeling environment the following logical predicates are automatically generated:

- `state(`p`,`s`)`: s is a state in automaton p.
- `npstate(`p`,`s`)`: s is a non-preemptable state in automaton p.
- `init(`p`,`s`)`: s is an initial state in automaton p.
- `in(`p`,`a`)`, `out(`p`,`a`)`, `internal(`p`,`a`)`: a is an input, output or internal action in automaton p, respectively.
- `step(`p`,`s`,`a`,`t`)`: there is a step in automaton p from state s to state t on action a.

The generated predicates capture the same information that is described in the graphical model. The advantage of logic program statements is undeniable after translating our definitions to logic program rules.

To demonstrate the effectiveness of this approach the Prolog version of Definition 3 is given:

```
% The intersection of internals(p) and actions(q) is empty
internal_fault(P,Q) :- internal(P,A), action(Q,A).
internal_fault(P,Q) :- internal(Q,A), action(P,A).
% The intersection of outs(p) and outs(q) is empty
out_fault(P,Q) :- out(P,A), out(Q,A).
% The intersection of ins(p) and ins(q) is empty
in_fault(P,Q) :- in(P,A), in(Q,A).
% Final rule
compose_fault(P,Q) :- internal_fault(P,Q).
compose_fault(P,Q) :- out_fault(P,Q).
compose_fault(P,Q) :- in_fault(P,Q).
```

After merging the translated rules and the automatically generated predicates in a *Prolog* [15] interpreter, one can check whether two automata p and q are compatible by asking the following question:

```
?- compose_fault(p,a).
```

The remaining definitions can be easily mapped as well; the entire toolset for compatibility checking consists of few lines of logic program code and a trivial model translator.

The current implementation applies basic reachability analysis on the composite automata by searching for illegal state configurations. One of the main benefits of this approach is the extremely simple implementation of the model translator. The relatively primitive structure of TinyOS interfaces allowed us to utilize this method. However, when more complex designs have to be verified or other intricate properties must be checked, employing a model checker might be a more appropriate choice. For these reasons, we have implemented a different translator for GRATIS recently, which targets SPIN [5], a well known verification tool.

Each automaton in GRATIS is mapped to a PROMELA *proctype* by transforming automata actions to *send* and *receive* statements and state transitions to flow control elements. For each pair of components separate *channels* are generated. The built-in rule for detecting invalid end states in SPIN will find all reachable illegal state configurations in the design. Other interesting properties—like liveliness or arbitrary assertions—can also be checked by labeling certain states. More sophisticated requirements can be formalized in linear time temporal logic (LTL) or by using *never* claims in the PROMELA language.

SPIN was created for detecting conceptual defects in distributed reactive systems. Considering the characteristics of our domain and comparing SPIN to our original approach, we should note the following difficulties:

- Interacting TinyOS components are not scheduled asynchronously, therefore, SPIN's *proctype* feature seems to be an overkill here.
- Our definition of compatiblity can easily handle open compositions (i.e. component configurations with unbound input and output actions). All unbound actions must be manually closed during model extraction, since SPIN is unable to handle open systems. This makes the GRATIS model translator for SPIN more complex.

6 MODEL VERIFICATION

Interface automata capture only the surface of software components, hence these descriptions are not sufficient for automatic code synthesis or simulation. They can be constructed even after the implementation phase of the component, as it is the case when dealing with existing TinyOS modules. Therefore, some kind of automatic verification is needed assuring consistency between the formal model and the implementation of components.

Model verification with existing source code—especially if the code was not created with a model centric approach—is a cumbersome if not impossible task. Analyzing sources written in traditional procedural languages, such as C, implies heuristic methods. Their dependency on the target implementation language makes the effort hard to justify.

We have chosen a different approach, which "interrogates" the existing modules and instead of trying to understand the source files, it generates additional code based on the interface models. The generated software behaves like a wrapper around the component to be tested, it generates each signal which is accepted by the component and it is prepared to receive all of the events coming from the module. The wrapper code reckons with the specified order of events, it executes the interface automata by transmitting actions in proper states and catching unexpected incoming messages. Figure 9 demonstrates the model verification process. This simple, yet powerful approach treats existing TinyOS modules as black box components, therefore, it can handle even the most obfuscated source code. Although this approach is not adequate for exploring the entire state space of an arbitrary software component, we succeeded in discovering interdependencies of interface primitives in real life TinyOS components.

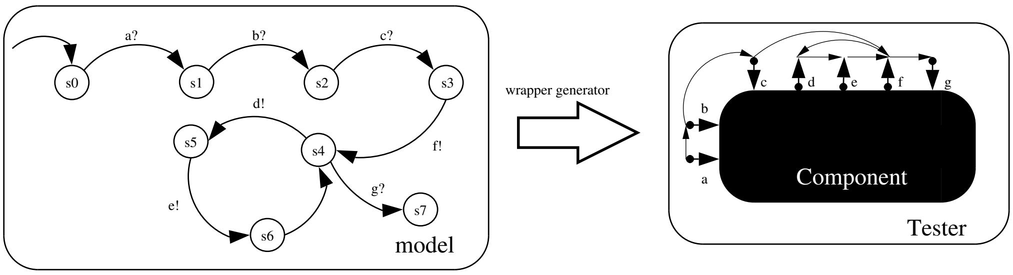

Fig. 9. The model verification process.

The prototype implementation of our black box testing approach targets the TinyOS platform. It requires manual intervention in those interface states where alternative output steps can be made by the tester, though automatic exhaustive testing could be easily achieved, as well. This tool has been integrated into the Gratis environment.

7 CASE STUDY

To demonstrate the expressiveness of the hierarchical interface automata language and the benefits of automatic composition checks, we further refine the visual model of the Sensor application introduced in Figure 2. In addition

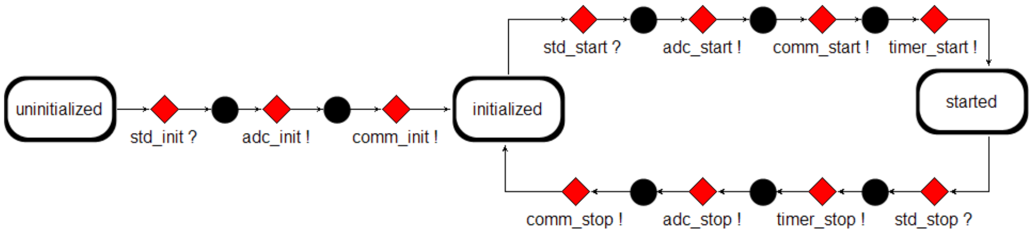

Fig. 10. Top level model of the *Sensor* component. Non-preemptable states are used extensively.

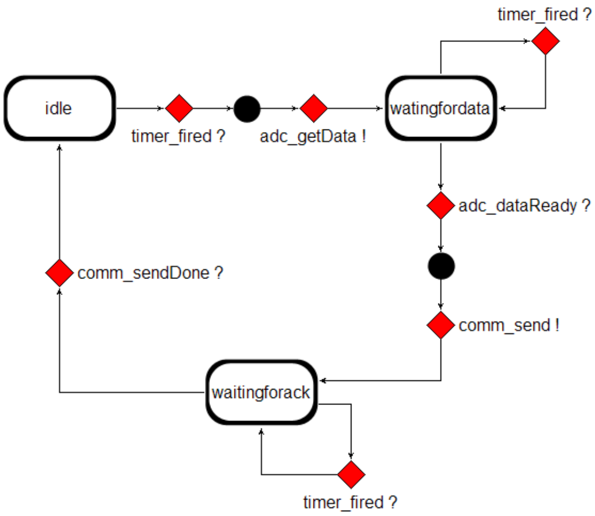

Fig. 11. Model of the *Sensor* component's busy loop.

to presenting compatible temporal models of the central and communication components, an alternative implementation will be shown where a small design flaw renders the application unreliable.

The top level model of the *Sensor* component is given in Figure 10. The lengthy action sequences perform initalization, startup and stop procedures. Upon receiving initialization/startup/stop requests the sensor component initializes/starts/stops the lower components subsequently. Note how non-preemptive states, denoted by filled circles, prevent the model from growing complicated and unreadable. The *started* state is refined in Figure 11. Without the use of hierarchical modeling, the *std_stop* action would have to be handled separately in each sub-state of *started*. In the *started* state the automaton repeatedly waits for timer events, requests data from the A/D converter, sends samples through the communication channel and then waits for the message buffer to be cleared.

The temporal model of the corresponding communication component is shown in Figure 13. Albeit only the inner service loop is shown, the model presents the restriction of the communication stack clearly: it is not prepared to process multiple messages simultaneously.

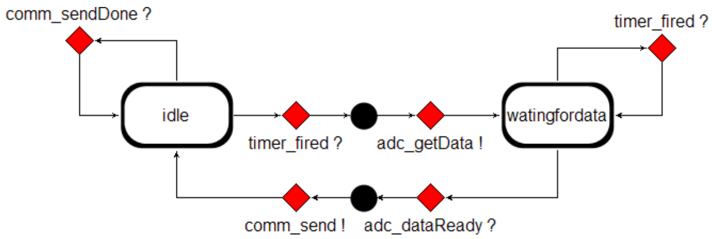

Fig. 12. Model of the faulty *Sensor* component's busy loop.

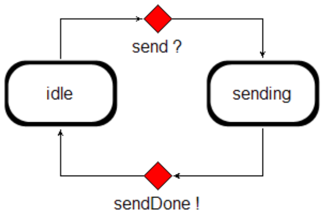

Fig. 13. Model of the *Comm* component's service loop.

The sensor component overcomes this limitation by waiting for a *sendDone* event before completing the iteration, thereby facilitating trivial flow control in the system.

The erroneous implementation of the sensor component shown in Figure 12 differs exactly in this regard. The automaton depicts a typical mistake; it essentially discards an event coming from the communication component. After its first iteration the sensor component may acquire a new sample from the A/D module, while the communication component is still in its *sending* state, where the *comm_send* event is not accepted. This application is unreliable, its operation depends on the timing properties of the data acquisition, periodic timer and task scheduling. This error—a reachable illegal state—is caught by the automatic verification tool. In contrast, manual debugging of similar problems may easily become a time consuming task.

8 CONCLUSIONS

The presented model-based approach to the component-based development of sensor network applications places special emphasis on interface specification. The proposed formalism captures the temporal and type aspects of interfaces and supports composition and verification of components. The implementation of the prototype modeling environment and the corresponding verification tools provided valueable feedback and influenced the design of the representation methodology.

The sensor example clearly demonstrated the benefits of our extensions to the traditional interface automata language, namely the hierachical representation of states and the introduction of non-preemptable conditions. Compatiblity checks with logic programs—although unconventional—prove to be extremely simple and straightforward to implement, ensuring consistency with the formal definitions. The presented extended Gratis environment significantly enhanced our TinyOS application development capabilities.

The nature of communication between components through function calls requires future study, since it does not fit the automata model perfectly: return values and constraints inherent in sequential flow of control are not captured by the current language.

Our current approach of compatibility checking suffers from scalability issues; the composition of n components requires $O(n^2)$ checks among these components. This is not a serious limitation considering the complexity of typical TinyOS applications, however it might prove to be a real problem in modeling of entire sensor networks.

References

[1] L. de Alfaro, T.A. Henzinger, Interface Automata, Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE), ACM Press, 2001, pp. 109-120.

[2] D. Harel, Statecharts: A Visual Formalism for Complex Systems, Science of Computer Programming, Elsevier, Vol. 8, No. 3, 1987, pages 231–274.

[3] A. Chakrabarti, L. de Alfaro, T.A. Henzinger, M. Jurdzinski, and F.Y.C. Mang, Interface compatibility checking for software modules, Proceedings of the 14th International Conference on Computer-Aided Verification (CAV), LNCS Vol 2404, Springer-Verlag, 2002, pages 428-441.

[4] K. Schneider, Verification of Reactive Systems, Formal Methods and Algorithms, Springer-Verlag, 2004.

[5] G. J. Holzmann, The SPIN Model Checker, Primer and Reference Manual, Addison-Wesley, 2003.

[6] E. M. Clarke, Jr., O. Grumberg, D. A. Peled, Model Checking, MIT Press, 2000.

[7] E. Mikk, Y. Lakhnech, and M. Siegel, Hierarchical automata as model for statecharts, Asian Computing Science Conference (ASIAN'97), Vol 1345 of LNCS. Springer-Verlag, 1997.

[8] N. Sabadini, R. Walters, Hierarchical Automata and P-systems, Electronic Notes in Theoretical Computer Science, Vol 78, Elsevier, 2003.

[9] Ji Wang, Wei Dong and Zhi-Chang Qi, Slicing Hierarchical Automata for Model Checking UML Statecharts, Proceedings of the 4th International Conference on Formal Engineering Methods, Springer-Verlag, 2002, pages 435–446

[10] A. Speck, E. Pulvermüller, M. Jerger, B. Franczyk, Component Composition Validation, International Journal of Applied Mathematics and Computer Science, Vol 12, pages 581–589, 2002.

[11] J. Hill at al, System Architecture Directions for Networked Sensors, Proceedings of ASPLOS, 2000.

[12] Á. Lédeczi, Á. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, G. Karsai, Composing Domain-Specific Design Environments, IEEE Computer 34(11), pp. 44-51, November, 2001.

[13] P. Völgyesi, Á. Lédeczi, Component-Based Development of Networked Embedded Applications, 28th EUROMICRO Conference (EUROMICRO 2002), Dortmund, Germany, September 4-6, 2002.

[14] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer and D. Culler, The nesC Language: A Holistic Approach to Networked Embedded Systems, Proceedings of Programming Language Design and Implementation (PLDI) 2003, June 2003.

[15] SICStus, SICStus Prolog User's Manual Swedish Institute of Computer Science, Sweden.

[16] S.J. Garland and N.A. Lynch, Using I/O automata for developing distributed systems, Foundations of Component-Based Systems, Cambridge University Press, 2000, pages 285–312.

[17] J. Warmer and A. Kleppe, The Object Constraint Language: Getting Your Models Ready for MDA, Second Edition, Addison-Wesley, 2003.