# Automatic Synthesis of Information Flow Driven Execution Managers for Embedded Software Applications

Nikita Visnevski,
*GE Research*
Niskayuna, NY, USA
nikita.visnevski@ge.com

Teresa Hubscher-Younger
*MathWorks*
Natick, MA, USA
thubsche@mathworks.com

Akshay Rajhans
*MathWorks*
Natick, MA, USA
arajhans@mathworks.com

Baoluo Meng
*GE Research*
Niskayuna, NY, USA
baoluo.meng@ge.com

***Abstract*—This paper presents an approach to simplifying embedded application development process by means of automatic synthesis of portions of application code concerned with information and execution flow management. In this preliminary study, we adopt a simple information flow categorization model which captures characteristics of data periodicity and criticality. This model is then used to define a formal specification of the interface of an embedded application. We show that, given such a specification, it is possible to automatically synthesize an application execution manager state machine that links the core of the application with various contingency management routines. This partially relieves the burden placed on embedded application developers allowing them to focus on development of the application core as well as on definition of actions to be taken when critical information flows are disrupted. They no longer need to create and manage code that monitors the information flows and implements logic for interrupting the execution of the core and invoking contingency management. This aspect would be automatically generated for them based on the design specification. Such automation offers a number of potential benefits to development productivity, cost reduction, software quality, validation, verification, and certification.**

***Index Terms*—embedded software, middleware, avionics, code generation**

## I. Introduction

Contemporary embedded software applications trend towards a high degree of modularization and utilization of abstraction layers to tightly encapsulate details of their implementation. They are expected to operate as standalone modules in real-time embedded deployment environments and be easily portable across multiple operating systems. The embedded deployment environments have evolved into complex, integrated platforms, offering scheduling, application management, and middleware-based information exchange services. It can thus be advantageous to develop embedded software applications following a "microservice" paradigm [1]. In doing so, however, application developers frequently encounter many challenges, some of which are not well understood, under-appreciated, and often addressed in an ad hoc fashion. This paper proposes an approach to addressing one such problem—abstracting core microservice algorithm implementation from the effects of asynchronous information flow. Our approach to algorithm synthesis and code generation yields a reliable application execution management structure resilient to adverse effects of asynchronous data transports.

Embedded applications consume information from or produce information for other applications as asynchronous messages using one or more middleware-based data transport services available on the deployment platform. They may have no control over the rate, regularity, and reliability of these messages, and need to incorporate contingency algorithms to deal with message dropout and communication anomalies. Traditionally, the implementation of the application is manually tailored to the set of possible anomalies in information flows in a highly application-specific fashion. As a result, the algorithmic details of execution and contingency management of the application become tangled with the core application code, yielding software that is difficult to maintain, evolve, verify, and certify. Application developers often put most of the effort into development of the nominal modes of execution of the algorithms. Abnormal conditions, however, typically result in most of the improperly handled application failures. Unfortunately, these abnormal conditions are rarely given enough attention at the application development stage. One of the reasons for it, as stated in [2], is simply that many embedded applications operating under nominal conditions are *transformative* in nature. Transformative systems are engineered based on well established theoretical foundations and formalisms. When anomalies caused by issues in distributed communication environment beyond the control of developers arise, the transformative system must instantly become *reactive*. Handling these reactive behaviors requires the use of very different theoretical formalisms. Thus, to fully cover all aspects of embedded system implementation, developers are forced to blend drastically distinct formalisms of both theoretical domains, which frequently proves to be highly problematic.

In this paper, we propose an alternative approach to addressing this problem. We break down an embedded software application into a core algorithm, a set of application-specific contingency management routines, and an execution management module that handles all aspects of middleware-based

information flows. The latter module orchestrates execution and contingency management policies of the application. Thus, the application core becomes completely decoupled from all adverse effects of asynchronous data transports. The state machine of the execution management module can be automatically synthesized based on a system-level specification of information flow types, namely mandatory vs. optional, and periodic vs. aperiodic. This state machine can be translated into executable code, which, together with the core algorithm, yields a fully-functional, resilient application.

To the best of our knowledge, this is the first time an automatic synthesis of this kind has been applied to the embedded software application development domain. The key benefit here is that embedded software developers can limit their efforts to development of the core algorithms and contingency management policies without the need to worry about ensuring proper data integrity, or deciding when to handle contingencies. This, in turn, yields much more concise, simple, manageable, maintainable, and verifiable software with promising potential to ease code certification.

This paper describes the problem in detail, presents an algorithm for execution manager synthesis, and works through a specific example developed in the Embedded Software integration Platform (ESiP) called Ensemble ESiP Toolbox. This specification-driven embedded software integration automation platform developed at GE Research is very well aligned with the model-based, specification-centric embedded software design philosophy of this paper.

## II. Problem Definition

In this section, we define the synthesis problem in concrete terms by introducing an illustrative example of a candidate embedded application—a nonlinear state estimator based on the Unscented Kalman Filtering (UKF) algorithm [3]. Such an estimator is often encountered in control, optimization, or diagnostics applications. It is particularly interesting to us as it relies on reception of time-sensitive data. We begin with a manual implementation of the UKF algorithm as a microservice, highlighting the prime areas for automation. We then restate the overall problem of this paper holistically and describe the assumptions made in solving it. This leads directly to the solution of this problem described in Section III.

### A. Illustrative Example

Imagine we are developing an embedded system responsible for solving a task of joint state and parameter estimation for a nonlinear system. A deployment scenario for this application could look like the illustration in Fig. 1 with systems as interface entities rendered in a variant of the Interface Definition Language (IDL) [4] binding for Unified Modeling Language (UML) [5] deployment diagrams.

An embedded deployment environment provides scheduling and message transport services. Systems are deployed as standalone microservices publishing and subscribing to message transport services thus forming distinct information flows across their interface boundaries. An estimator named `UKFMicroservice` is an embedded application subscribing to input information flows including measurement, initialization, and plant information messages. These messages originate from different data providers (an instrumentation module and settings and parameter database). The microservice is also a producer of estimate data and linearized model information for a controller module. Estimates are also fed back into a settings database as a basis for future estimator initializations (see Fig. 1).

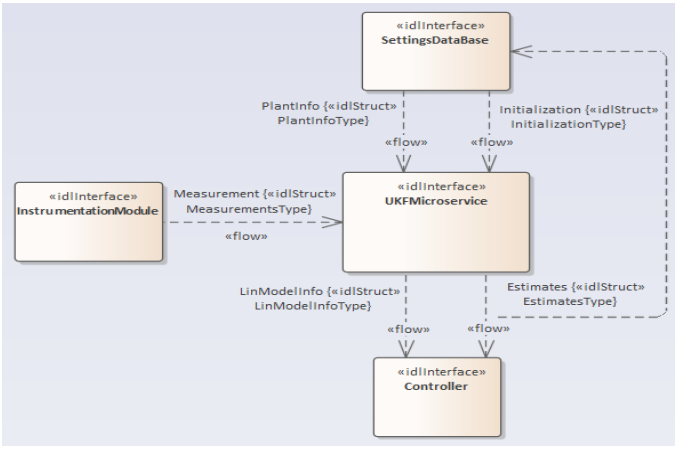


Fig. 1. System and execution context of an estimation microservice in an embedded deployment environment.

Using IDL binding for UML class diagrams, we can describe the various types of structured data that form the messages in the information flow model above (see Fig. 2). Readers familiar with recursive least squares estimation algorithms will easily recognize and relate to the data structures shown in Fig. 2. They represent the observation, estimation, and correlation data commonly used in Bayesian estimation approaches.[1] Those interested in more in-depth treatment of Bayesian estimation theory can refer to an excellent and very accessible tutorial paper [6].

High-level design in Fig. 1–2 can be implemented in a variety of different ways each of which has its own advantages and drawbacks. We make use of Model-Based Design in Simulink® [7] and automatic code generation using Embedded Coder® [8] to realize the implementation of the estimator microservice. A Simulink implementation of the estimator microservice core is shown in Fig. 3.

In this implementation, a generic UKF estimator, abstracted away from application-specific implementation details, is coupled with plant dynamics modeled as a Simulink function `model_dynamic_xKF` with a standardized interface. This pair of blocks interact with each other to produce an application-specific implementation of the UKF estimator. In addition to the UKF algorithm for state and parameter estimation, this core module also calculates plant-specific linear

[1] In addition to common data entries, this data model also includes some application-specific data structures (e.g., `ModelMetaDataType`, `PlantParamsType`). Though relevant to implementation, they are not pertinent to the discourse of this paper, and thus are omitted from the model.

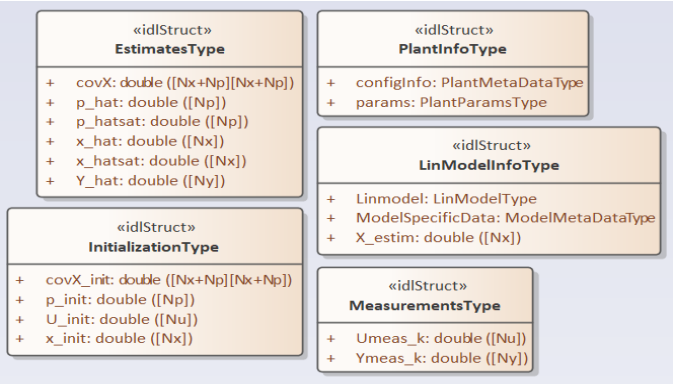


Fig. 2. Data type definitions for estimator microservice data flows.

models at estimated state and parameter values to be used by the optimizing controller, as illustrated in Fig. 1.

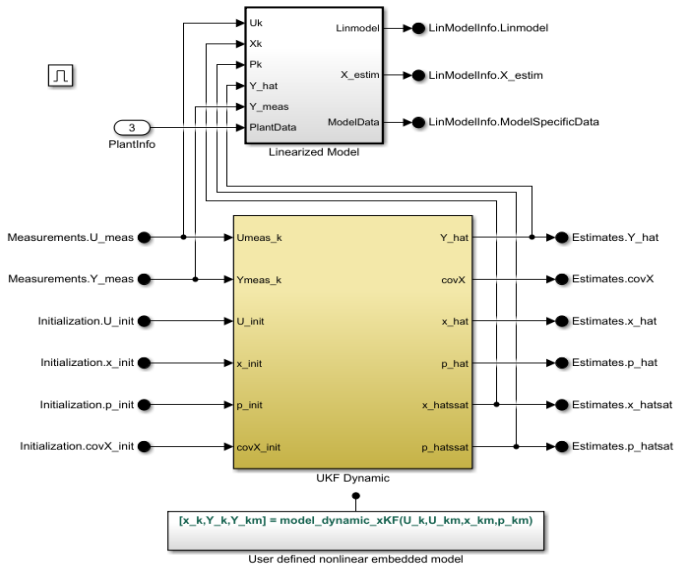


Fig. 3. Simulink® model of an estimator application core based on the Unscented Kalman Filter algorithm.

Having defined an implementation of the core application algorithm, we now need to add communication and execution management layers to the microservice model to complete its implementation. Here, as before, a great variety of approaches is possible, and, as before, we use Model-Based Design in Simulink to structure the microservice as shown in Fig. 4.

The UKF core model in Fig. 3 assumes a constant and reliable presence of properly time-correlated data as its inputs at all times, which cannot be guaranteed in distributed deployment environments. Thus, additional steps need to be taken to preserve input data integrity given the sensitivity of Bayesian estimators to coherency of input data. As seen in Fig. 4, inputs are received, and outputs are transmitted, as messages from/to middleware connectivity blocks commonly referred to as data subscribers and publishers. In addition to actual data required by the core estimation algorithms, middleware connectivity layers need to provide communication status and integrity metadata, which predictably encodes the statuses of transport layer connectivity and message reception for data publishers and subscribers.

Assuming availability of this additional information from the transport connectivity layer, we can build an execution manager algorithm that enables the execution of the UKF core only when input data integrity is assured, and that ensures transmission of output data only when it is appropriate. In our example we use a Stateflow® [9] chart to implement the algorithm of this execution manager (see Fig. 4).

A sample pseudo implementation of the execution manager algorithm is shown in Fig. 5. Microservice designers must be able to enable correct operation of the estimator under nominal conditions as well as properly handle all possible data transport-induced contingencies, such as absence of connectivity, absence of initialization information, and delays in measurement data. A more sophisticated implementation of this microservice will likely include additional intricacies, but for the purpose of our illustrative example it is quite adequate to identify these three.

The state machine in Fig. 5 has been simplified to show only the details relevant to the big picture. Once an application starts up, the execution manager needs to ensure that data transport connectivity is properly established. If not, a specific contingency management action must be taken. In this illustrative example, the chosen contingency management routine for all the contingencies simply waits for a timeout and terminates the application if the situation does not resolve itself before timeout occurs. In practice, a more sophisticated management routine may be implemented.

Once the connectivity is established, the execution manager needs to correctly initialize the UKF, as improper initialization may lead to a prohibitively long convergence time. Absence of initialization data is therefore viewed as another contingency our execution manager must manage. Otherwise, after correct initialization, we can advance the application state by executing the core with properly sampled input measurements. If a measurement message is not received, yet another contingency arises. We assume that the Stateflow implementation properly extracts information payloads from all messages and feeds correct data structures into the algorithm core block (this part is not shown on the pseudo implementation Fig. 5).

We must observe that it is critical, whether relying on Model-Based Design principles in Simulink or any other tool, or using any other design and implementation paradigm, to adhere to the two most fundamental principles of software engineering—*abstraction* and *encapsulation*. This enables a clear separation of concerns of individual system components, and lets the designer fix some implementation aspects, while varying others to improve the overall quality and robustness of the implementation. As we will see in Section III, this separation is fundamental to our approach. It allows us to introduce a considerable amount of automation and code synthesis of the state machine that represents the microservice execution manager. Such automation relieves embedded system develop-

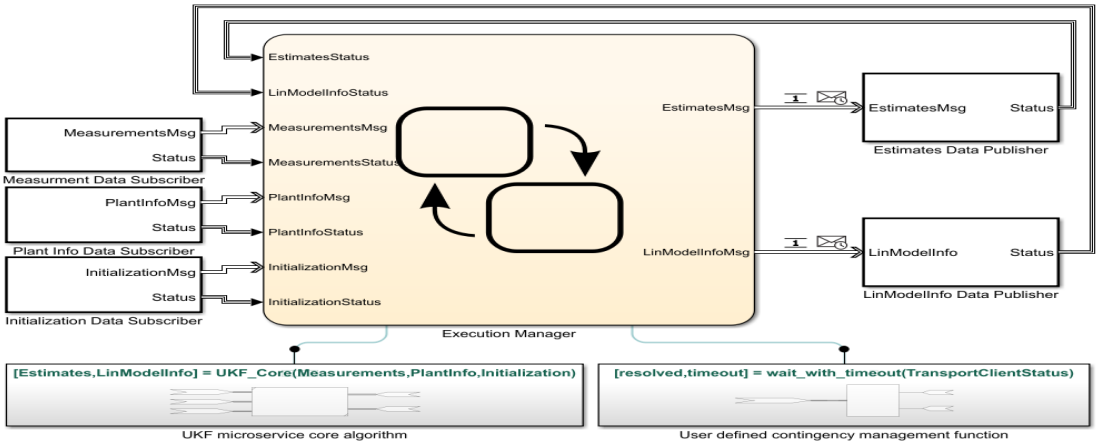


Fig. 4. Simulink® model of a microservice of an embedded estimator application based on Unscented Kalman Filter algorithm.

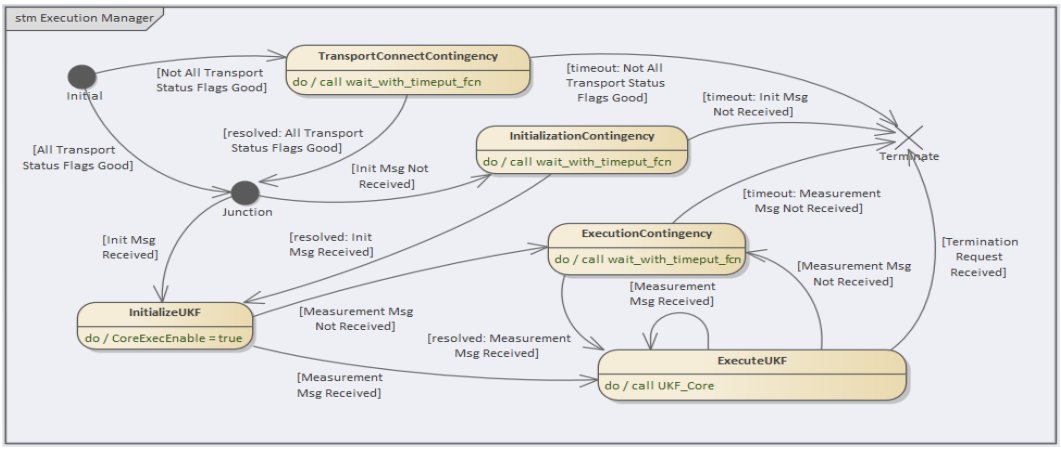


Fig. 5. Pseudo statechart representing a possible manual implementation of the microservice Execution Manager.

ers from the burden of developing complex logic for tracking status and integrity of input data. Developers need to only invest effort and time in developing the algorithmic core and handling any possible contingencies. A solid design combined with a clearly-defined specification of the microservice I/O structure will enable state machines such as the one shown in Fig. 5 to be generated fully automatically.

### *B. Problem Statement and Assumptions*

The illustrative example from Fig. 4 is now generalized into one of the fundamental postulates of our approach. Namely, that any embedded software application can be broken down into four mutually independent sets of components:

- a set of middleware connectivity modules (publishers and subscribers) for incoming and outgoing information flows of the application,
- a core algorithm implementation module,
- an execution manager state machine, and
- a set of contingency management routines tailored to specific issues that could arise in an asynchronous communication environment.

The interfaces of components in the above sets can be defined such that their implementations become fully encapsulated and independent from each other. In that case, an embedded application can be fully specified by three types of artifacts:

- an implementation of the application core algorithm,
- an implementation of the set of contingency management routines, and
- a formal application interface specification linking application information flows to corresponding contingency management routines.

The final integration of all components of the embedded application can then be fully automated using these three artifact types as a starting point. In this paper, we consider only one step of this automation—synthesis of the execution manager state machine using the application interface specification.

Our approach is based on the set of the following assumptions:

- Fixed time step algorithm execution—a single application code iteration as well as any execution and contingency management routines fully complete within a single ex-

ecution time step.

- Asynchronous information flow—processes governing information delivery to applications are independent of the application execution timing.
- Applications are responsible for determining incoming information validity, recency, timing relationships with the algorithm state, and actions to be taken in response to the information flow.
- Information flows are message-based. Messages received by the applications contain payload and metadata. Payload contains application-specific data. Metadata includes standardized information related to middleware connectivity status, quality of service, recency, validity, periodicity, etc.

The assumptions above define a specific framework within which the problem will be examined. Additionally, we assume that incoming information flows of an embedded application can be categorized as:

- periodic (expected to arrive at fixed, predetermined time intervals allowing for some deterministic or stochastic jitter) or aperiodic (expected to arrive at arbitrary points in time), and
- optional (application will execute normally even in the absence of these incoming messages) or mandatory (application cannot execute normally in the absence of the incoming messages).

These simplifying assumptions from this preliminary study would need to be relaxed in future work to include a more complete and thoroughly developed model of asynchronous, middleware-based information exchange, especially including various quality of service modes offered by many middleware implementations.

## III. Execution Manager Synthesis Algorithm

As described in Section II-B, we categorize the types of information flows produced and consumed by an embedded system as:

- **mandatory periodic**—flows in which information is expected to arrive and depart at specific time intervals and for which absence of information arrival and departure at expected point in time requires contingency management;
- **mandatory aperiodic**—flows that are expected to arrive and depart at least once in the lifetime of an application and absence of information from these flows requires contingency management;
- **optional periodic**—supplementary flows for an application that are expected to arrive in periodic sequences;
- **optional aperiodic**—same as previous, but no periodicity is implied.

For the last two categories of flows, contingency management requirements are typically narrowed in scope to simple notification of the status of the information flow and do not involve suspending application core execution process.

Each of these four categories of information flows can be associated with an elementary state machine addressing their respective requirements. These elementary state machines are illustrated in Fig. 6–9. Specifically, the mandatory periodic flow scenario can be handled by the state machine in Fig. 6. At the start of the application, the sate machine analyzes the transport connectivity information and verifies that the connectivity is valid, that is, the reception or transmission is occurring. If not, a user-defined contingency management routine is invoked. This routine may establish that the contingency has been properly addressed or may issue a timeout. If resolved, the transport status checks are repeated. If timed out, the application is terminated. If all status flags are in order and information flow is successful, a nominal execution state is entered where the application core can be executed. After each step of application core execution, transport status must be re-verified. In the context of our example from Section II-A, the measurement data stream for the UKF should be handled in this fashion.

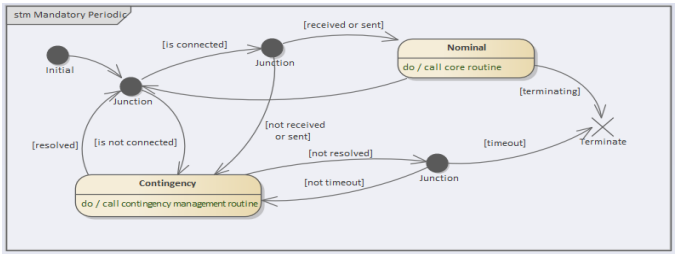


Fig. 6. Elementary state machine for mandatory periodic information flows.

The elementary state machine for the mandatory aperiodic flow type is shown in Fig. 7. It is fundamentally similar to Fig. 6 except that once the state machine enters its nominal operation state, it can stay there throughout the life cycle of the application. In the context of our UKF example, the initialization and plant info data streams fall into this category. The UKF algorithm must receive each at least once in order to proceed. It may receive them more than once, but at least once is required for operation.

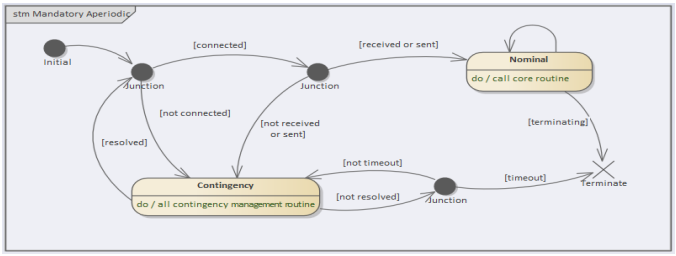


Fig. 7. Elementary state machine for mandatory aperiodic information flows.

Finally, execution managers for optional information flows can be modeled as state machines in Fig. 8–9. The same principles apply here except we do not define explicit contingency states and implement the calls to contingency management routines as part of the state machine transition logic. In the context of the UKF example we can view outputs of the system (`Estimates` and `LinModelInfo`) as optional periodic

information flows. The core algorithm generates this data at every execution step and publisher blocks make an attempt to publish these structures. If the publication is unsuccessful for some reason (e.g., because of connectivity or communication problems), appropriate status values may be fed back into the state machine. The expected behavior is up to the application designer. Here, for example, we continue application execution calling our contingency management routine as a notification callback every time a transmission problem occurs. This mode of operation fits the optional periodic model.

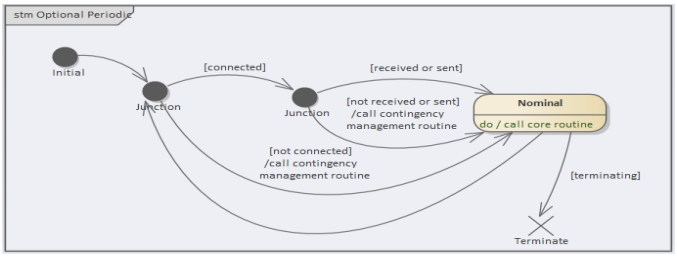


Fig. 8. Elementary state machine for optional periodic information flows.

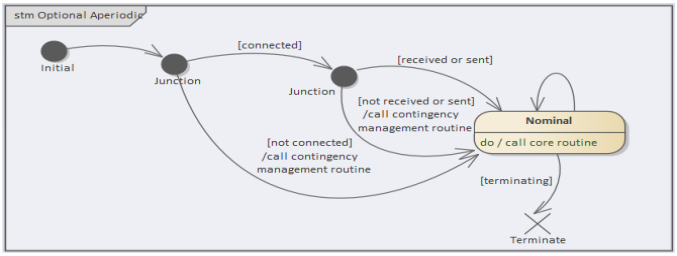


Fig. 9. Elementary state machine for optional aperiodic information flows.

We postulate that a holistic solution to a problem of execution manager synthesis for an embedded application is essentially the problem of composing a set of elementary state machines covering specifications of all incoming and outgoing information flows of that application. Here we propose and notionally describe such an algorithm. In Section IV, we show how an execution manager for our UKF microservice example can be synthesized by this algorithm.

The state machines in Fig. 5–9 have been defined using the classical Moore/Mealy automata formalisms (see [10], [11]).[2] It is possible to derive the algorithm that combines elementary state machines associated with application information flows into a single sequential state machine handling all necessary cases. It is, however, rather impractical to do so since in general such a state machine may contain a very large number of states for applications with many information flows. The modern Harel statechart framework [2] combines Mealy and Moore formalisms and adds parallelism in execution. It also allows for parallel state machines to dynamically exchange information. Using the Harel statechart framework, the algorithm for synthesizing a complete execution manager can be simply expressed as a three-step process:

1) combining all elementary state machines for all information flows in a single statechart with parallel decomposition,
2) extracting the core algorithm execution step into a separate concurrent statechart, and
3) establishing conditional logic and an event structure for allowing the two resulting statecharts to interoperate.

Section IV provides a detailed illustration of how this notional algorithm description can be used to synthesize an execution manager for the UKF estimation microservice.

## IV. Estimation Microservice Revisited

We now show how to apply the automated synthesis concept from Section III to the example embedded application described in Section II-A. We will examine this state machine synthesis task in the context of a much larger design automation framework. This frameworks aims at the following:

- enabling rapid and effective ways to design, integrate, test and deploy embedded software,
- maximizing development productivity using specification-based model synthesis and automatic code generation, and
- establishing a process workflow for reinforcing developmental discipline and team productivity via rigorous specification modeling, Model-Based Design as well as early and frequent software and system integration.

We first briefly introduce this framework and provide a specification of the UKF estimator microservice to be utilized by the framework. We then present and discuss the execution manager state machine for our example application as it will be generated by the framework.

### A. Brief Introduction to Ensembele ESiP Toolbox

Ensemble ESiP Toolbox is a tool developed by GE Research to drastically increase productivity of embedded software development teams through maximum use of automation. The tool showcases the following innovations:

- It combines development, test, integration, and deployment into a single, cohesive, high-productivity workflow.
- It offers the ability to deploy software in a modularized format on heterogeneous, multi-platform deployment environments.
- It introduces a new declarative specification language for embedded software and system definition. The toolbox uses system specifications in this format to automate model and code generation for all software components, alleviating the need for re-implementing known and well-established functionality.
- It establishes a flexible information exchange infrastructure interface using message API allowing embedded applications to be easily configured to use a variety of interchangeable and extensible data transports.

[2]Strictly speaking, in Fig. 8–9 we intentionally blend Mealy and Moore notations by allowing state machine actions as part of state transition logic to maintain compactness.

The typical workflow of the toolbox is conceptually illustrated in Fig. 10. The development team is expected to supply a detailed system specification describing what embedded software applications constitute the system definition. In addition, build and deployment configuration of the system is specified. It includes, among other things, detailed data type definitions of all information flows accross all component boundaries as well as types of middleware utilized for deployment. The development team is also expected to provide implementations of application-specific core algorithmic components as well as all application-specific contingency management routines. The Embedded Software Component Factory processes all these input artifacts, matches I/O specifications with available middleware and data transport libraries and generates a fully integrated system test harness as well as deployable microservice models such as the one illustrated in Fig. 4.

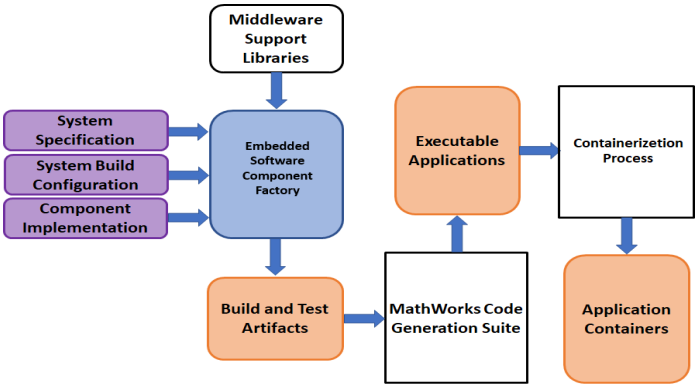


Fig. 10. Ensemble ESiP Toolbox simplified workflow.

Auto-generated build and test artifacts serve as inputs to the automated code generation suite, which builds and integrates the microservice models into target platform specific executables. An optional configuration step allows the toolchain to containerize the applications for deployment in container-based deployment environments. Next, we show how this toolchain can be applied to synthesis of the UKF estimation microservice defined in Section II-A.

### *B. Synthesis of Estimation Microservice*

To synthesize a UKF estimation microservice with the Ensemble ESiP Toolbox, we need to develop a data type definition specification for all information flows illustrated in Fig. 1. The UML model of this specification is shown in Fig. 2 and Ensemble ESiP Toolbox provides a utility to import UML models in a standard XMI format and convert them to an IDL data type definition per specification [4]. For our example, such a data type definition specification looks as follows:

```
struct PlantInfoType {
    PlantMetaDataType   configInfo;
    PlantParamsType     params;
};
struct LinModelInfoType {
    LinModelType        Linmodel;
    ModelMetaDataType   ModelSpecificData;
    double              X_estim[Nx];
};
struct EstimatesType {
    double  covX[Nx + Np][Nx + Np];
    double  p_hat[Np];
    double  p_hatsat[Np];
    double  x_hat[Nx];
    double  x_hatsat[Nx];
    double  Y_hat[Ny];
};
struct InitializationType {
    double  covX_init[Nx + Np][Nx + Np];
    double  p_init[Np];
    double  U_init[Nu];
    double  x_init[Nx];
};
struct MeasurementsType {
    double  Umeas_k[Nu];
    double  Ymeas_k[Ny];
};
```

The second necessary artifact is the system and build configuration specification. Ensemble ESiP accepts this specification in multiple formats (XML, JSON, YAML, MATLAB® code, XMI, and others). A simplified version of this specification in YAML is as follows:

```
system:
    name: UKFDeploymentEnvironment
    modules:
        - name: InstrumentationModule
          ioDescriptors:
            - name: Measurement
              ioType: OUTPUT
              idlDataType: MeasurementsType
        - name: SettingsDataBase
          ioDescriptors:
            - name: PlantInfo
              ioType: OUTPUT
              idlDataType: PlantInfoType
            - name: Initialization
              ioType: OUTPUT
              idlDataType: PlantInfoType
            - name: PlantInfo
              ioType: INPUT
              idlDataType: PlantInfoType
        - name: Controller
          ioDescriptors:
            - name: LinModelInfo
              ioType: INPUT
              idlDataType: LinModelInfoType
            - name: Estimates
              ioType: INPUT
              idlDataType: EstimatesTypeType
        - name: UKFMicroservice
          srcName: UKFMicroserviceImpl.mdl
          buildConfig:
            srcFormat: SIMULINK_MDL
            contingencyManagers:
                - contingencyManager:
                    name: wait_with_timeout
                    srcName: wait_with_timeout.m
                    buildConfig:
                        srcFormat: MATLAB_FCN
                  io:
                    - Measurement
                    - Initialization
                    - PlantInfo
                    - LinModelInfo
                    - Estimates
          ioDescriptors:
            - name: Measurement
              ioType: INPUT
              idlDataType: MeasurementsType
              optional: false
              periodic: true
            - name: Initialization
              ioType: INPUT
              idlDataType: PlantInfoType
              optional: false
              periodic: false
            - name: PlantInfo
              ioType: INPUT
```

```
    idlDataType: PlantInfoType
    optional: false
    periodic: false
  - name: LinModelInfo
    ioType: OUTPUT
    idlDataType: LinModelInfoType
    optional: true
    periodic: true
  - name: Estimates
    ioType: OUTPUT
    idlDataType: EstimatesTypeType
    optional: true
    periodic: true
```

The spec above is basically an extension and generalization of the system and execution context definition from Fig. 1. It defines a `UKFDeploymentEnvironment` system consisting of four independently deployable application modules. The first three, namely `InstrumentationModule`, `SettingsDataBase`, and `Controller` serve to define execution, test, and evaluation contexts of our main module named `UKFMicroservice`. Specification of the modules serving as the execution context are defined in an abbreviated form, whereas the definition of the `UKFMicroservice` module itself is provided in more detail.

The core algorithm of the UKF microservice is provided by a Simulink model `UKFMicroserviceImpl.mdl` shown in Fig. 3. Specification of information flows is given in the `ioDescriptors` section. It includes three inputs and two outputs. For simplicity, we omit deployment configuration parameters that specify middleware bindings for each IO flow. Ensemble ESiP Toolbox uses RTI DDS [12] as a default transport if explicit specification is omitted. Each IO descriptor defines what category the IO flow belongs to. As one can see, we have one mandatory periodic, two mandatory aperiodic, and two optional periodic flow definitions. Contingency managers are defined in a separate portion of the module build configuration. We define a single contingency manager routine associated with all five information flows. In practice, however, one may define separate contingency managers for inputs and outputs, or for each individual category of information flows. The contingency manager is a MATLAB function in the file `wait_with_timeout.m`. The implementation of this function is omitted here, as it is expected that the developer will provide it.

The four artifacts described above serve as input artifacts for the Embedded Software Component Factory of the Ensemble ESiP Toolbox shown in Fig. 10. It produces, among other things, a fully functional UKF microservice model with correct middleware connectivity bindings, execution manager structures, and linkage to algorithm core and contingency management routines. Fig. 4 shows the result of this automatic synthesis process.

The resulting execution manager state chart structure is shown in Fig. 11. Synthesis of this Stateflow chart follows the algorithm notionally defined in Section III. Individual state machines handling specific IO flows are combined together through parallel decomposition in a single `ExecManager` chart. `MeasurementManager` is a detailed realization of the notional definition of the state machine in Fig. 6. `InitializationManager` and `PlantInfoManager` implement the state machine in Fig. 7. Finally, `EstimatesManager` and `LinModelManager` implement the state machine shown in Fig. 8.

Each state machine in the `ExecManager` chart monitors the status and connectivity information of its information flow and is responsible for deciding if the application should be in nominal or contingency mode. Statechart `CoreExector` is a result of steps 2 and 3 in the algorithm from Section III. It consists of only the functionality of calling the core routine, separated from all the other logic of execution management. It implements simple voting logic on the nominal state flags from `ExecManager`, transitions into the `ExecCore` state when appropriate, calls the core algorithm routine to compute current estimates of the UKF, sends messages with output payloads out to data publishers, and sends a notification event back to `ExecManager` to resume information processing. Figure 11 captures the state machine in action in its nominal state executing UKF core function.

## V. Conclusion

This paper presents a preliminary study of one of the steps in automating embedded software application design, development, and deployment. This step is responsible for correctly and robustly handling anomalies induced by the nature and complexities of distributed communication in the application execution context. We show that, to some extent, the difficulties of synthesizing a *reactive* embedded application around a *transformative* core can be addressed through good understanding of the nature of the deployment environment, competent design, and automation. To that end, this paper presented and analyzed a framework that offers potential solution to this problem. Despite the many simplifying assumptions in our reasoning and arguments, it is our hope that this idea is viewed as promising and that the software community considers building upon it to further improve and simplify the development process.

One obvious area of improvement would be to augment the approach we described with existing middleware Quality of Service (QoS) models. This may be a nontrivial task in general as QoS models vary greatly across different middleware implementations. Another area of improvement would include a critical look at our information flow categorization model presented in Section III. A more complete and accurate model can be derived by refining the quality of the final execution manager structure. Another improvement would involve studying the cross-coupling effects of different contingencies and understanding how they may affect each other in a simultaneous multiple-fault scenario. Yet another area of improvement might include structural optimization of the execution managers in order to potentially reduce the number of states and transitions in the final state machine and to simplify the logic. Finally, a big challenge would be to examine verification, validation, and safety certification aspects of the code generated using this approach.

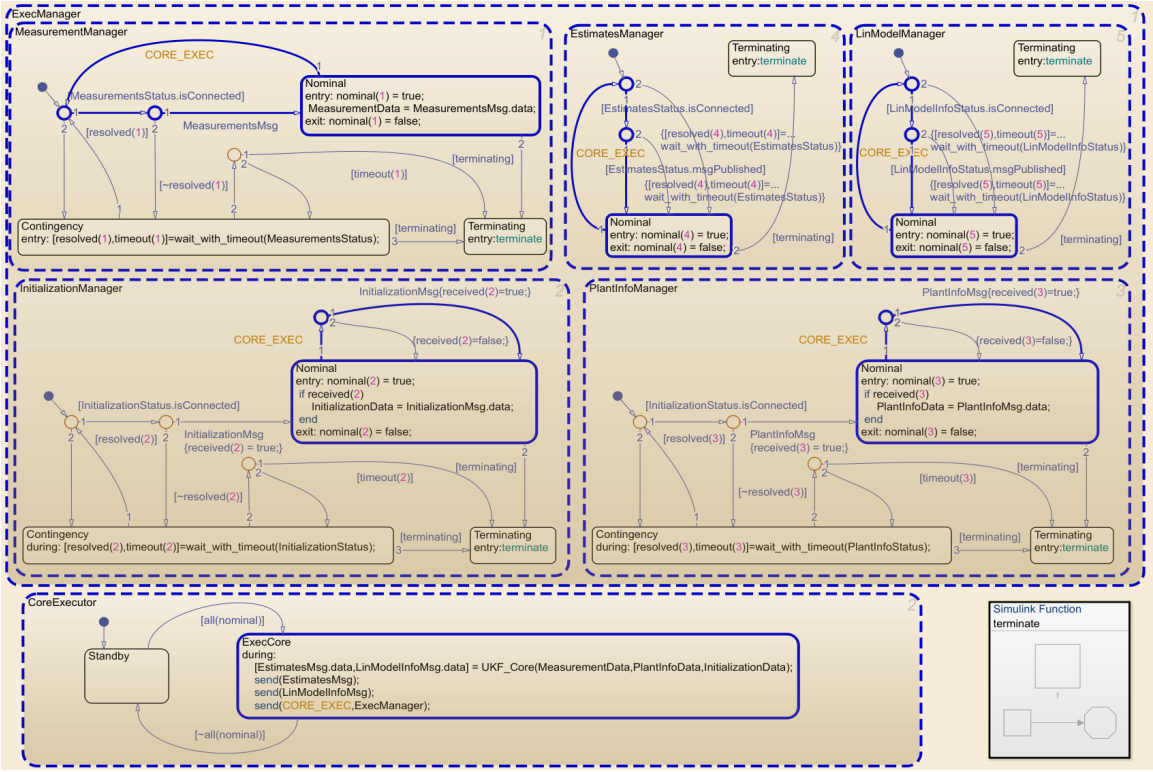


Fig. 11. UKF example execution manager statechart synthesized based on the developed algorithm.

Over the past few decades, development of software, especially embedded software, has dramatically increased in complexity and cost. Developers are often expected to have both depth and breadth of knowledge in a wide variety of areas of theory and practice in order to succeed in their product design and deployment. Often this burden proves to be too much. The work presented here aims at finding creative and competent ways of using automation to help development teams keep up and stay in their primary domain of expertise, while automating as many aspects of the development and deployment processes as can possibly be automated. This, in our opinion, is the main contribution of this paper.

## Acknowledgment

Authors would like to thank Dr. Aditya Kumar and Dr. Mustafa Dokucu of GE Research for sharing the Unscented Kalman Filtering estimation core shown in Fig. 3 and extracted from GE Research Estimation and Controls design library.

## References

[1] X. Larrucea, I. Santamaria, R. Colomo-Palacios, and C. Ebert, "Microservices," *IEEE Software*, vol. 35, no. 3, pp. 96–100, 2018.

[2] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[3] E. A. Wan and R. Van Der Merwe, "The Unscented Kalman Filter for nonlinear estimation," in *Proceedings of the IEEE 2000 Adaptive Systems for Signal Processing, Communications, and Control Symposium*, 2000, pp. 153–158.

[4] "Interface Definition Language (IDL) version 4.2," Object Management Group (OMG), Standard, Mar. 2018. [Online]. Available: https://www.omg.org/spec/IDL/4.2

[5] "Unified Modeling Language (UML) version 2.5.1," Object Management Group (OMG), Standard, Dec. 2017. [Online]. Available: https://www.omg.org/spec/UML/2.5.1

[6] A. H. Sayed and T. Kailath, "A state-space approach to adaptive RLS filtering," *IEEE Signal Processing Magazine*, vol. 11, no. 3, pp. 18–60, 1994.

[7] MathWorks®, "Simulink® R2019b," September 2019. [Online]. Available: https://www.mathworks.com/products/simulink.html

[8] MathWorks®, "Embedded Coder® R2019b," September 2019. [Online]. Available: https://www.mathworks.com/products/embedded-coder.html

[9] MathWorks®, "Stateflow® R2019b," September 2019. [Online]. Available: https://www.mathworks.com/products/stateflow.html

[10] E. F. Moore, "Gedanken-experiments on sequential machines," in *Automata Studies, Annals of Mathematical Studies*. Princeton, NJ: Princeton University Press, 1956, vol. 34, p. 129–153.

[11] G. H. Mealy, "A method for synthesizing sequential circuits," *Bell System Technical Journal*, vol. 34, pp. 1045–1079, 1955.

[12] Real-Time Innovations (RTI), "Connext DDS 6," April 2019. [Online]. Available: https://www.rti.com/products