Chapter 3
Synoptic: A Domain-Specific Modeling Language for Space On-board Application Software


3.1 Introduction

3.1.1 Context: Embedded Flight Software

This section describes the context of the SPaCIFY project and the various constraints that have been handled in the design and implementation of the Synoptic language toolset.
Satellite and Flight Software

A satellite is an unmanned spacecraft. The system architecture is usually specialized according to the satellite mission. There are two main subsystems: the payload and the platform. The payload consists of mechanical structure, sensors and actuators used by the payload and devices for communications with ground stations. The SPaCIFY project focuses on the flight software embedded in the satellite to manage its platform, also called on-board software. The flight software is a real-time software that provides services that are common to whatever mission-specific payload the spacecraft is assigned. Typical services include reaching and following the desired attitude and orbit, managing thermal regulation systems, power sources, monitoring system status, managing on-board network (MIL-STD-1553, OBDH, SpaceWire), and communicating with ground stations.

Even after the satellite has been launched, flight software must be adapted. Satellites are subject to high energy particles that may damage hardware components. Such damages cannot be fixed except by installing software workarounds. Bug fixes and software updates should be propagated to flying satellites. In addition, mission extensions may require functional enhancements.

As flight software is critical to the success of the mission, space industries and agencies have worked on engineering processes in order to help increase reliability. For instance, the European Space Agency has published standards (ECSS-E-40) on software engineering and (ECSS-Q-ST-80) on product assurance. These standards do not prescribe a specific process. They rather formalize documents, list requirements of the process and assign responsibilities to involved partners. Regarding updates, standards stipulate for instance that the type, scope and criticality must be documented; that updates must be validated; and so on. Industries are free to come up with their own conforming processes. The SPaCIFY project defines such a process and supporting tools based on Model-Driven Engineering (MDE), synchronous languages and the Globally Asynchronous Locally Synchronous System (GALS) paradigm.

Kind of Model Used

Two main kinds of models are commonly used in order to design the platform software: on the one hand, the description of the platform itself, its hardware and software architecture, CPU, memory, storage, communication buses, sensors, actuators, hardware communication facilities, operating system, tasks, software communication facilities, and on the other hand, the command and control algorithms involved in the management of the platform. All these models have common features. First, the notion of mode is central to represent the various states or configurations of both hardware and software (for example: init, reset, on, low power, failure, safe). The modes and their management are usually expressed using finite automata. Second, the functional blocks, data exchange buses and signals are used to represent both the hardware and software architectures, and the command and control software.
However, the current models only provide a partial account of the constraints that the final system must satisfy. The designers usually encode these hidden constraints using the available constructs in the modeling languages, even if this was not the initial purpose of the construct used. This is usually the case for hard real-time constraints. The designers will rely on explicit management of the control-flow in dataflow models in order to manage the concurrency between the activities. But, the real timing constraints are not explicitly given in the model, they are handled by the designers who sequence the activities, but there is no real specification of the intended result. Thus it requires a very difficult verification phase that can only occur on the final target. This is the current main difficulty: using model constructs for other purpose than their intended ones without any formal, model-level, traceability to the initial constraints.

Models in Development Process

Currently, industrial main actors are relying on the Matlab toolboxes Simulink and Stateflow from “The MathWorks” [4] for expressing the command and control algorithms for the various sub-systems of the platform. These models are built by command and control engineers taking into account several kinds of constraints:

- The hardware which is usually known very early (in particular if it handles timing and synchronisation constraints related to sensors and actuators, which must be activated and will produce results in very precise timing patterns)
- The system mode management that impacts the execution of the various sub-systems
- The specification of the intended sub-system

The Simulink and Stateflow toolboxes allow a very wide range of modeling methods, from continuous partial differential equations to graphical functional specifications. Each industrial actor has a well defined process which defines the restricted subset of the modeling language that will be used at each phase of the development cycle. In the ITEA GeneAuto project [34], a subset of these toolboxes was defined that fits the needs for the modeling of space application from early design to automated target code generation. The industrial partners of SPaCIFY took also part in GeneAuto. This subset aimed at providing a solid semantic background that would ease the understanding and the formal verification of models. This subset was chosen as an entry point for the design of the Synoptic language. One key point is the use of control-flow signals and events (called in Simulink function call events) in order to manage explicitly the sequencing of the blocks in dataflow models. This is an important point which is significantly different on the semantics side from the classical dataflow modeling languages such as SCADE [16] or RT-Builder [3] which do not provide a specific modeling construct for this purpose, but allow to encode it through empty data signals that are added between each block in the intended sequencing path. The key point is that the intended hard real-time constraints are not explicit. Thus it is mandatory to handle the control-flow construct exactly in its usual semantics not using an approximate encoding which only works most of the time but is not proven to work in all cases.
Several studies have been conducted by industrial main actors regarding the modeling of hardware and software architecture. HOOD [30] and CCM [20] have been used for many real projects; AADL [6, 31], SysML [18] and UML/MARTE [29] have been evaluated using already existing projects that could be modeled and the results compared with the real systems. Once again, these languages provide a very wide range of modeling constructs that must be reduced or organized in order to be manageable. In the European ASSERT project [5], two tracks were experimented related to these languages, one mainly synchronous based on the LUSTRE [23] and SIGNAL [22] languages, the other mainly asynchronous based on the RAVENSCAR Ada [13] profile. The industrial partners from SPaCIFY were also part of ASSERT. Thus, the results of these experiments were used as entry points for the design of the Synoptic language.

In the current software development process, command and control models are formal specifications for the software development. These specifications have been validated by command and control engineers by using model simulators. The hardware architecture is currently defined in a semi-formal manner through structured documents. In the near future, models in AADL, or in a subset of SysML/UML/MARTE similar to AADL, will be used to give a formal specification of the architecture.

Then, software engineers either develop the software and verify its conformance to the specification, or use automated code generators; the software is then split in parts that are mapped to threads from the RTOS. They are then scheduled according to the real-time constraints. The know-how of engineers lies in finding the best splitting, mapping and scheduling in order to minimize the resources used.

One of the purposes of introducing Model Driven Engineering is to be able to automate partly these manual transformations and the verification that their result satisfies the specification. The Synoptic language should thus allow to import command and control models expressed in Simulink/Stateflow, hardware architecture models expressed in AADL. The associated toolset should assist in the re-organisation of the model, and allow to express the mapping and scheduling.

### 3.1.2 Domain Specific Requirements

#### Real Time Constraints

Flight software is mainly responsible for implementing command and control laws. It is therefore a set of tasks performed at fixed time periods by a real-time kernel. Tasks perform the various management activities of the system. They have severe time constraints inherited from the command and control design by automation. The number of tasks varies depending on the approach adopted by the industry: Thales Alenia Space favors a large number of tasks (about 20–30 active tasks on a total of 40–50); while EADS Astrium has a tendency to aggregate calculations, including frequency activation to reduce the number of tasks.
Although the constraints are strong, the time scale is relatively slow. The activation periods of tasks in the flight software vary typically between 100 ms and 100 s.

**Limited Hardware Resources Capacity**

The computing resources embedded in satellites are limited. Thus, at the time of writing, the memory allocated to the flight software is generally under 10 MB. The binary image is also stored in a 1–2 MB EEPROM.

The computing power ranges from the 20 MIPS ERC32 processor up to nearly 100 MIPS for the LEON 2 and 3 ones. Moreover, satellites operate in a hostile physical environment. Shocks, temperature variations, radiation, high energy particle beams, damage and destroy electronic components, leading to rapid aging of embedded equipments.

**Remote Observability and Commandability**

Once a satellite has been launched, the only interventions currently possible are remote. Several scenarios have been identified that require maintenance interventions such as for example:

- The satellite, and therefore its embedded software, must survive as long as possible to equipment aging and damages. Installing software workarounds is an economical means of continuing the mission as long as remaining equipments permit. Identifying problems is critical for the ground engineering teams that design such workarounds.
- Satellites are often still usable at the end of their initial mission. Updating the software is an economical way of achieving new mission objectives while recycling satellites. Therefore, ground operators must adapt the software to changing operational conditions.
- When a bug is detected in the software, its correction should be remotely installed without compromising the satellite.

These scenarios emphasize the need for remote monitoring and management of the flight software. Monitoring probes characteristics and state of satellite components (both hardware and software) and the physical environment. Management provides actions both at the application level, e.g., manoeuvre, and at the middleware level, e.g., software update.

The communication link between satellites and ground passes through ground stations. Since the SPaCIFY project has removed from its study the possibility of routing communications through other spacecrafts, being in range of a ground station is a prerequisite for connectivity. Characteristics (speed, visibility window)
of the communication link depend of the orbit and of the mission. In general, we
consider that it is rarely possible to download or upload a full image of the flight
software.

A High Level of Quality and Safety

Given the fact that the only possible actions once the satellite has been launched
are performed remotely, it is essential to ensure that the satellite is always able
to respond to maintenance requests. In addition to verifying the software, defensive
mechanisms are usually added as for example to restart computers in case of trouble.
Indeed, given the radiation to which satellites are exposed during flight, high-energy
particles can swap bits in memory. Several images of the software are also stored on
board to be sure to have at least a version which is sufficiently functional to enable
communications with the ground.

A Particular Software Industry

In addition to software business constraints, the design of flight software has its own
peculiarities.

For certain classes of satellites, including scientific satellites, each satellite is
a single unit, in fact a prototype. Indeed, every scientific mission has a specific
objective, needing specific measuring equipment.

For other categories, including communications satellites, they are built in se-
ries from a single design. However, due to the unpredictability of last minute client
requests, manufacture, launch and space environment hazards, each satellite soon
becomes again unique. This inevitably leads to a specific flight software for each
different satellite. Even when multiple identical copies are initially installed on sev-
eral satellites, such copies diverge to take into account the specific situation of each
satellite and especially their failures and specific maintenance.

Furthermore, it is difficult to perform realistic testing of flight software. At best,
simulations can give an indication of the software correctness.

GALS Systems

A satellite management software is usually divided in parts that are quite au-
tonomous one from the other, even if they share the same platform and resources.
Inside each of these parts, the subparts are most of the time strongly linked and
must cooperate in a synchronous manner. These parts usually exchange informa-
tion but with less time critical constraints, thus relying on asynchronous exchanges.
These kind of system are usually called Globally Asynchronous, Locally Syn-
chronous. Synoptic must provide some specific modelling elements to handle this
structure.
3 Synoptic: A DSML for Space On-board Application Software

### 3.1.3 Synoptic: A Domain Specific Design Environment

#### Motivations

In collaboration with major European manufacturers, the SPaCIFY project (The SPaCIFY Consortium 2008) aims at bringing advances in MDE to the satellite flight software industry. It focuses on software development and maintenance phases of satellite lifecycles. The project advocates a top-down approach built on a domain-specific modeling language named Synoptic. In line with previous approaches to real-time modeling such as Statecharts and Simulink, Synoptic features hierarchical decomposition in synchronous block diagrams and state machines. SPaCIFY also emphasizes verification, validation and code generation.

One key point to ease the long-term availability and the adaptability of the tools, is to rely on open-source technologies.

#### Overview of the SPaCIFY Development Process

To take into account the domain specific requirements and constraints presented in the previous section, the SPaCIFY project proposes to introduce models inside the design and maintenance processes of the on-board applications. Thus, the space domain may benefit from technologies developed in the context of Model Driven Engineering for processing, handling and analyzing models. It therefore takes a shift from current practices based on documents to a tool-supported process built around formal models.

The Fig. 3.1 summarizes the major steps of the proposed development and design process. The central models are described in the Synoptic domain specific and

![Fig. 3.1 Sketch of the SPaCIFY development cycle for on-board software](image-url)
formal language defined by the project. The process promotes vertical and horizontal transformations. The vertical transformations sketch the refinements and the enrichments of Synoptic models. The horizontal transformations consist of translation transformations to formal models (Altarica [1], SME [9] models) that are equipped with specific verification, code generation or simulation tools.

The initial Synoptic model, resulting from the automated transformation of Simulink/Stateflow models previously designed by control engineers, is enriched by non-functional properties derived from textual requirements.

Using the hardware specifications, the Synoptic model is further enriched by the hardware model of the system. Based on hardware and software constraints, the dynamic architecture can be derived.

Finally, the resulting model includes the different views of the system (software, hardware, dynamic architectures) and mappings between them. This last model is used to generate the source code and to configure the middleware of the embedded application. At each step, analysis and verifications can be performed. The transformations of models used in the refinement process formalize the expertise acquired by the industry.

Figure 3.2 focuses on the code generation phase. At this phase, the model of the application is initially translated into a SME model. Code generation itself is performed by the Polychrony compiler [21]. The generated code targets the use of a middleware specifically designed for the SPaCIFY project.

Contents of the Chapter

This chapter is organized as follows. Section 3.2 introduces the main features of the Synoptic DSML with a particular focus on the functional sub-language of Synoptic, called Synoptic core. Synoptic core permits to model synchronous islands. They communicate through asynchronous shared variables (called *external variables*) managed by the middleware. A simplified case study is used to present the various concepts introduced. Section 3.3 describes the formal semantics and the polychronous model of computation of the Synoptic core which is based on the synchronous dataflow language SIGNAL [7]. Section 3.4 focuses on middleware aspects. The architecture of the execution platform, the middleware kernel, external variables, and the reconfiguration service of the middleware are discussed. Section 3.5 concludes the main contribution of the project and gives an outlook on future investigations.
3 Synoptic: A DSML for Space On-board Application Software

3.2 Synoptic: A Domain Specific Modeling Language for Aerospace Systems

3.2.1 Synoptic Overview

Synoptic is a Domain Specific Modeling Language (DSML) which aims to support all aspects of embedded flight-software design. As such, Synoptic consists of heterogeneous modeling and programming principles defined in collaboration with the industrial partners and end users of the SPoCIFY project.

Used as the central modeling language of the SPoCIFY model driven engineering process, Synoptic allows to describe different layers of abstraction: at the highest level, the software architecture models the functional decomposition of the flight software. This is mapped to a dynamic architecture which defines the thread structure of the software. It consists of a set of threads, where each thread is characterized by properties such as its frequency, priority and activation pattern (periodic, sporadic).

At the lowest level, the hardware architecture permits to define devices (processors, sensors, actuators, busses) and their properties.

Synoptic permits to describe three types of mappings between these layers:

- Mappings which define a correspondence between the software and the dynamic architecture, by specifying which blocks are executed by which threads
- Mappings which describe the correspondences between the dynamic and hardware architecture, by specifying which threads are executed by which processor
- Mappings which describe a correspondence between the software and hardware architecture, by specifying which data is transported by which bus for instance

Figure 3.3 depicts the principles discussed above.
Our aim is to synthesize as much of these mappings as possible, for example by appealing to internal or external schedulers. However, to allow for human intervention, it is possible to give a fine-grained mapping, thus overriding or bypassing machine-generated schedules. Anyway, consistency of the resulting dynamic architecture is verified by the SPaCiFy tool suite, based on the properties of the software and dynamic model.

At each step of the development process, it is also useful to model different abstraction levels of the system under design inside a same layer (functional, dynamic or hardware architecture). Synoptic offers this capability by providing an incremental design framework and refinement features. To summarize, Synoptic deals with dataflow diagrams, mode automata, blocks, components, dynamic and hardware architecture, mapping and timing.

In this section we focus on the functional part of the Synoptic language which permits to model software architecture. This sub-language is well adapted to model synchronous islands and to specify interaction points between these islands and the middleware platform using the concept of external variables. Synchronous islands and middleware form a Globally Asynchronous and Locally Synchronous (GALS) system.

Software Architecture

The development of the Synoptic software architecture language has been tightly coordinated with the definition of the GeneAuto language [34]. Synoptic uses essentially two types of modules, called blocks in Synoptic, which can be mutually nested: dataflow diagrams and mode automata. Nesting favors a hierarchical design and allows to view the description at different levels of details.

By embedding blocks in the states of state machines, one can elegantly model operational modes: each state represents a mode, and transitions correspond to mode changes. In each mode, the system may be composed of other sub-blocks or have different connection patterns among components. Apart from structural and behavioral aspects, the Synoptic software architecture language allows to define temporal properties of blocks. For instance, a block can be parameterized with a frequency and a worst case execution time which are taken into account in the mapping onto the dynamic architecture.

Synoptic has a formal semantics, defined in terms of the synchronous language SIGNAL [9]. On the one hand, this allows for neat integration of verification environments for ascertaining properties of the system under development. On the other hand, a formal semantics makes it possible to encode the meta-model in a proof assistant. In this sense, Synoptic will profit from the formal correctness proof and subsequent certification of a code generator that is under way in the GeneAuto project. Synoptic is equipped with an assertion language that allows to state desired properties of the model under development. We are mainly interested in properties that permit to express, for example, coherence of the modes ("if component X is
in mode m1, then component Y is in mode m2" or "\( \ldots \) can eventually move into mode m2\). Specific transformations extract these properties and pass them to the verification tools.

3.2.2 Software Architecture Models

One typical case study under investigation in the project is a generic satellite positioning software, Fig. 3.4. It is responsible for automatically moving the satellite into a correct position before starting interaction with the ground.

The specification of the central flight software (OBSW), consists of the composition of heterogeneous diagrams. Each diagram represents one specific aspect of the software’s role. The OBSW module is controlled by a remote control telecommand TC001 and receives attitude and position data (POS\_Data and ATT\_Data) from sensors through the middleware.

The Attitude and Orbit Control System (AOCS) is the main sub-module of the central flight software, Fig. 3.5b. The AOCS automaton controls its operational modes for specific conditions. It is either in nominal mode or in SAFE mode, Fig. 3.5c. The safe mode is characterized by a rude pointing of satellite equipments (solar panels, antennas). It computes the command to be sent to the thrusters to maintain a correct position and attitude.

3.2.2.1 Block Diagram: Dataflow, Automaton and External Blocks

A Synoptic model is a graphical block-diagram. A Synoptic block-diagram is a hierarchy of nested blocks. As such, you can navigate through different levels of abstraction and see increasing levels of model details. A block is a functional unit that communicates with other blocks through its interface.

---

Fig. 3.4 Satellite positioning software
Fig. 3.5 A typical case study: satellite positioning software
A block interface consists of communication ports. A port can be either an event port or a data port and is characterized by its direction (in or out). Data ports can be typed using simple data types. However, typing data ports is optional in the early stages of design to give the designer the flexibility to describe abstract models.

As shown in Fig. 3.6, which represents a part of the Synoptic metamodel, it is possible to encapsulate a set of ports within a single entity called group of ports (PortGroup in Fig. 3.6). A group of ports is used to group a set of ports which are conceptually linked. As specified by the synchronized property of the PortGroupDecl meta-model class, it is possible to specify a synchronization constraint on ports constituting the group.

A block interface can be implemented by different types of blocks: dataflows, automata or externals, Fig. 3.7.

**Dataflow**

A dataflow block models a dataflow diagram. It embodies sub-blocks and specifies data or event flows between them. A flow is a directed connection between ports of sub-blocks. A sub-block is either an instance of dataflow, automaton or external.
Fig. 3.7 Synoptic metamodel: model diagram

block, or an instance of block interface (see ModelInstance class in Fig. 3.7). In the latter case, the model is abstract: the designer must implement all the block interfaces used to type sub-blocks in order to obtain a concrete model. As such, Synoptic environment design promotes a top-down approach.

Automaton

An automaton block models state machines. A Synoptic automaton consists of states and transitions. As for dataflow, a state can be represented by an instance of dataflow, automaton or block interface. Dataflow diagrams and automata can be hierarchically nested; this allows for a compact modelling of operational modes.

Apart from the ongoing actions of automata (block embedded in state), it is possible to specify entry and exit actions in an imperative style. Transitions between states are guarded and equipped with actions. There are two types of transitions in Synoptic: strong and weak transitions. Strong transitions are used to compute the current state of the automaton (before entering the state), whereas weak transitions are used to compute the state for the next instant.

More precisely, the guards of weak transitions are evaluated to estimate the state for the next instant, and the guards of strong transitions whose source is the state estimated at the previous instant, are evaluated to determine the current state.

External Block

A common practice in software industry is the re-use of external source code: designers must be able to introduce blocks representing existing source code into the model. Moreover, for modeling flight software functional units, it is necessary to use primitive operations such as addition, cosinus, division.
Synoptic: A DSML for Space On-board Application Software

Synoptic provides the concept of *external block* to model primitive blocks and external source code, Fig. 3.7. Primitive blocks are encapsulated in a Synoptic library. The Synoptic tool suit recognizes these primitive operations during code generation.

For embedding an external source code, the procedure is quite different. The designer must build his own external block by defining its interface, by giving the source code path which will be used at code generation time, and by specifying pre/post conditions and the Worst Case Execution Time of the functional unit.

**Example: A Dataflow Block Diagram**

Figure 3.8 shows the graphical decomposition of the nominal mode of the Attitude and Orbit Control System (AOCS/NM) depicted in (Fig. 3.5b). The header of the block tells us that NM is an instance of the NM_{diff} dataflow block. In nominal mode, AOCS can either use its sun pointing sensors (SUP) to define its position or, during eclipse, use its geocentric pointing sensors (GAP). To model this activity, the nominal mode of the AOCS flight software encapsulates two sub-blocks: one dataflow block (TEST_{ECLIPSE}) which detects an eclipse occurrence and one automaton block (SUP_{OR}_GAP) which ensures the change of sub-mode (SUP or GAP) depending on the outcome of the eclipse test block.

3.2.2.2 Synchronous Islands

A functional model consists of synchronous islands. A synchronous island is a synchronous functional unit in interaction with the middleware. The middleware supports the execution of code generated from the Synoptic model: it provides a synchronous abstraction of the asynchronous real world.

![Fig. 3.8 AOCS nominal (NM) sub-mode – satellite positioning software](this figure will be printed in bw)
To ensure proper integration of the generated code with the middleware platform, it is essential to locate all interactions between the middleware and the application. Our approach is to locate all these interactions into a single concept in Synoptic: the concept of external variables. External variables allow to represent in Synoptic models the following domain specific and technical concepts: telecommand, telemetry, constants database systems, shared variables and persistent variables (on reboot). The specification of external variables are used for code generation and middleware configuration.

Therefore a Synoptic synchronous island is a block in which all input and output signals are connected to external variables. The OBSW block of our case study is one such synchronous island, Fig. 3.5a.

**Contractual Approach**

An external variable is a variable managed by the middleware. It can be read or written by multiple clients (components of the application). Contracts are used to specify how to control this access.

The configuration of an external variable is done using four levels of contracts:

1. A classical **syntactic contract** that includes its name and type
2. A **remote access contract** (through telemetry)
3. A **persistence contract** that specifies if the variable is persistent on reboot or not
4. A **synchronization contract** that describes the possible concurrent interactions when the variable is shared

Each component using such a variable must also provide a **usage contract** that defines the protocol it will use to access the variable.

These contracts allow the middleware to manage the variable. Thus, the synchronization contract will indicate if locks are needed and when they should be used. The persistence contract will indicate how the variable should be stored (RAM or permanent memory). The monitoring functions of the variable that the middleware must implement are defined by the remote access contract and the usage contracts.

**3.2.2.3 Control and Temporal Properties**

As described before, blocks are functional units of compilation and execution. Block execution is controlled using two specific control ports: trigger and reset ports. These ports are inherited from the Simulink/Stateflow approach [4]. These ports appear as black triangles in the upper left of the block, Figs. 3.5 and 3.8.

**Trigger Port**

The trigger port is an event port. The occurrence of a trigger event starts the execution of the block and its specification may then operate at its own pace until the next trigger is signaled.
3 Synoptic: A DSML for Space On-board Application Software

Sub-blocks have their own trigger control ports and can thus operate at a different rate. Without event signal connecting its control port, a sub-block inherits the control signals of its parent block.

**Explicit Clock and Adaptors**

The trigger port stands for the clock of the block. In our case study, the trigger control port of blocks is never connected: the clock of the block is explicitly defined using a period or a frequency property. Adding this frequency property is semantically equivalent to connecting the trigger control port with a 20-Hz clock.

By convention, all input signals of the block are synchronized with the trigger of the parent block. This constraint may be too strict: it is thus possible to redefine the frequency of ports by adding an explicit property of frequency. Note however that the clock of ports must be a down-sampling of the parent trigger.

Explicitly specifying a real-time constraint on ports and blocks can lead to difficulties when specifying a flow between two ports with different frequencies. **Synoptic** tools are able to detect such clock errors. The designer should use pre-defined *clock adaptors* to resample the signal, Fig. 3.8.

Besides frequency properties, it is possible to specify the phase and the Worst Case Execution Time (WCET) of a block.

**Reset Port**

The reset port is a boolean data port whose clock is a down-sampling of the trigger signal. The reset signal forces the block B to reset its state and variables to initial values.

Contrarily to StateCharts [24], when a transition is fired the destination state of the transition is not reset. It means that by default, one enters the history state. To reset the destination state and all variables it contains, the solution is to specify this reset in the action of the transition.

**3.2.2.4 Properties Specification**

The Synoptic language is equipped with an assertion language to state desired properties of the model under development. This language makes it possible to express invariants, pre- and post-conditions on blocks.

Invariants are used to express coherence of modes. For instance, a typical assertion that we want to prove on the case study model (Fig. 3.5) is that if the MCS block is in SAFE mode, then the AOCS block is also in SAFE mode. Such an assertion is described in Synoptic as follows:

$$(\text{OBSW.MCS.state} = \text{SAFE}) \implies (\text{OBSW.AOCS.state} = \text{SAFE})$$
The Synoptic environment provides a tool to extract Synoptic models and their associated properties and pass them to the Altarica model-checker [1]. Pre- and post-conditions can be either statically or dynamically tested. In the latter case, monitoring functions are implemented in the final software and raise events when properties are not satisfied.

Monitoring functions are distinguished from assertion properties by raising an event to its environment when the property is not satisfied:

```
Assertion : pre : a < 10 ;
Monitoring function : pre : a < 10 raise e! ;
```

Here, \( a \) stand for an input data port and \( e \) for an event port and \( e! \) for an event emission on \( e \) port.

### 3.2.3 Synoptic Components: Genericity and Modularity

Synoptic supports modular system development and parametric components. The latter are particularly useful for an incremental design and gradual refinement, using predefined development patterns (see Sect. 3.2.4).

As mentioned in Sect. 3.2.2.1, there are two main categories of components: dataflow blocks and automata blocks. They exist on the type level (interfaces) and on the element level (instances). Components can be parametric, in the sense that a block can take other elements as arguments. Parametric components are similar to functors in ML-style programming languages, but parameters are not limited to be blocks. They can, among others:

- Be integers, thus allowing for variable-sized arrays whose length is only fixed during compile time
- Be types, thus allowing for type genericity as in Java 5 [19]
- Be entire components, thus allowing for component assembly from more elementary blocks

Syntactically, the parameters of a block are specified after a `requires` clause, the publicly visible elements made available by the block follow in a `provides` clause, and there may be a `private` part, as in Fig. 3.9.

This example reveals parts of the textual syntax of Synoptic, which in this case is more perspicuous than a graphical syntax. The component \( C \) requires (an instance of ) a block, called \( \text{ADD} \), that is characterized by a component type having a certain number of in and out ports. \( C \) provides a dataflow that is composed of two blocks, one of which is defined in the private part of \( C \).

Parameterized blocks can be instantiated with elements that meet their typing constraints, in a large sense. In the case of simple parameter types (like integers), the typing constrains are evident. When instantiating a parameterized component having a parameter type \( P \) with a component \( C \), the component \( C \) has to provide all the elements stipulated by the `requires` clause of \( P \) (but may provide more).
3 Synoptic: A DSML for Space On-board Application Software

```plaintext
component C
  requires
  block type ADD
  features
    idp1 : in data port integer;
    idp2 : in data port integer;
    odp1 : out data port integer;
  end ADD;
  provides
    dataflow ADD_MULT.op
    blocks
      add : block type ADD;
      mult : block type MULT;
    flows
      s1 : data idp1 \rightarrow add.idp1;
      \quad-- other flows ...
    end ADD_MULT.op;
  private
  block type MULT
  features
    \quad-- ...
  end MULT;
end C;
```

Fig. 3.9 A parameterized component

Conversely, \( C \) may require some (but not necessarily all) the elements provided by \( P \). Parameter instantiation is thus essentially contravariant. Some clauses of a component are not checked during instantiation, such as `private`.

Parameter types can be equipped with properties (see Sect. 3.2.2.4) such as temporal properties. Instantiating these types gives rise to proof obligations, depending on the underlying logic. In some cases, an exact match between the formal parameter component and the actual argument component is not required. In this case, a mechanism comparable to a type cast comes into play: Take, for example, the case of a component triggered at 20 Hz \( C_{20j} \) (as in Fig. 3.8) that is to be used by a parametric component \( C_{10j} \) operating at 10 Hz. Component \( C_{20j} \) can be used as argument of \( C_{10j} \), and a default frequency splitter will be synthesized that adapts the frequency of the \( C_{20j} \) to \( C_{10j} \).

3.2.4 Incremental Design and Refinement Features

The Synoptic environment promotes a top-down approach including incremental design and refinement features.
In first steps of design, Synoptic allows to describe an abstract model. For instance, the designer can describe abstract interfaces where data ports are not typed and connect instances of these interfaces in a dataflow diagram.

In a second step, the designer can refine its modeling by typing the block interfaces. The block interface can be then implemented with dataflow or automaton blocks. These features of the Synoptic environment are mainly "edition refinement" which allows the designer to model a system in an incremental way. In doing so, the designer loses the link between the different levels of refinement: when the model is concretized, the initial abstract model is not preserved and therefore cannot be accessed.

Synoptic offers a way to specify refinement and to keep a formal link between an abstract model and its concretization. As depicted in the metamodel, Synoptic provides two types of refinements: flow refinement and block refinement, Fig. 3.10.

A flow refinement consists of refining a flow with a dataflow block. To be properly defined, the flow must be declared as abstract and the dataflow block must have a single input port and a single output port correctly typed.

A block refinement consists of refining a block instance with a dataflow block. The main use of this type of refinement is to concretize an interface block with a dataflow.

Example

Figure 3.11 illustrates a flow refinement. The first model is an abstract view of the global architecture of the case study.

The abstract model consists of three dataflow blocks. Sensors send their data (position and attitude) to the central flight software. OBSW computes them and sends a new command to actuators. In this abstract block diagram, flows between actuators and the OBSW are obviously not synchronous signals: data are exchanged through an 1553 bus.
To consider this fact, flows are specified as abstract. The same applies to the flows between OBSW and actuators. Moreover, a real-time requirement is added to the flows: $\ll \text{delay} \leq 20 \mu s \gg$.

This requirement represents the maximum age of data: data consumed by OBSW must be aged less than 20 ms. The second model in Fig. 3.11 shows how this abstract model is refined. Each flow is refined with a dataflow block composed of external variables.

The concrete model confirms that data sent by the sensors are not Synchronous flows but pass through the middleware. According to the real-time requirement of abstract model, the middleware is in charge of distributing data to flight software by respecting the limit of data aging. In addition, this refinement step displays the OBSW synchronous island.

3.3 Semantic and Model of Computation of synoptic Models

The model of computation on which Synoptic relies is that of the Eclipse-based synchronous modeling environment SME [9]. SME is used to transform Synoptic diagrams and executable generate C code. The core of SME is based on the synchronous dataflow language SIGNAL [22]. This section describes how Synoptic programs are interpreted and compiled into this core language.

3.3.1 An Introduction to SIGNAL

In SIGNAL, a process $P$ consists of the composition of simultaneous equations $x = f(y, z)$ over signals $x, y, z$. A delay equation $x = y \text{prev}$ defines $x$ every time $y$ is present. Initially, $x$ is defined by the value $v$, and then, it is defined by the
previous value of $y$. A sampling equation $x = y$ when $z$ defines $x$ by $y$ when $z$ is true. Finally, a merge equation $x = y$ default $z$ defines $x$ by $y$ when $y$ is present and by $z$ otherwise. An equation $x = yfz$ can use a boolean or arithmetic operator $f$ to define all of the $n$th values of the signal $x$ by the result of the application of $f$ to the $n$th values of the signals $y$ and $z$. The synchronous composition of processes $P | Q$ consists of the simultaneous solution of the equations in $P$ and in $Q$. It is commutative and associative. The process $P / x$ restricts the signal $x$ to the lexical scope of $P$.

$$P, Q ::= x = yfz | P / x | P | Q$$  

(process).

In SIGNAL, the presence of a value along a signal $x$ is an expression noted $x$. It is true when $x$ is present. Otherwise, it is false. Specific processes and operators are defined in SIGNAL to manipulate clocks explicitly. We only use the simplest one, $x\ sync y$, that synchronizes all occurrences of the signals $x$ and $y$.

### 3.3.2 Interpretation of Blocks

**Blocks** are the main structuring elements of Synoptic. A block $xA$ defines a functional unit of compilation and of execution that can be called from many contexts and with different modes in the system under design. A block $x$ encapsulates a functionality $A$ that may consist of sub-blocks, automata and dataflows. A block $x$ is implicitly associated with two signals $x:\ trigger$ and $x:\ reset$. The signal $x:\ trigger$ starts the execution of $A$. The specification $A$ may then operate at its own pace until the next $x:\ trigger$ is signaled. The signal $x:\ reset$ is delivered to $x$ at some $x:\ trigger$ and forces $A$ to reset its state and variables to initial values.

$$(blocks) \quad A, B ::= blockxA | dataflowxA | automatonxA | A | B.$$  

The execution of a block is driven by the trigger $t$ of its parent block. The block resynchronizes with that trigger when itself or one of its sub-blocks makes an explicit reference to time (e.g., a skip for an action or a delayed transition $S \rightarrow T$ for an automaton). Otherwise, the elapse of time is sensed from outside the block, whose operations (e.g., on $c$), are perceived as belonging to the same period as within $[t_i, t_{i+1}]$. The interpretation implements this feature by encoding actions and automata using static single assignment. As a result, and from within a block, every non-time-consuming sequence of actions $A; B$ or transitions $A \rightarrow B$ defines the value of all its variables once and defines intermediate ones in the flow of its execution.

### 3.3.3 Interpretation of Dataflow

**Dataflows** inter-connect blocks with data and events (e.g., trigger and reset signals). A flow can simply define a connection from an event $x$ to an event $y$, written
event \( x \rightarrow y \), combine data \( y \) and \( z \) by a simple operation \( f \) to form the flow \( x \), written data \( y \circ f \circ z \rightarrow x \) or feed a signal \( y \) back to \( x \), written data \( y \text{ pre} \rightarrow x \).

In a feedback loop, the signal \( x \) is initially defined by \( x_0 = v \). Then, at each occurrence \( n > 0 \) of the signal \( y \), it takes its previous value \( x_{n-1} \). The execution of a dataflow is controlled by its parent clock. A dataflow simultaneously executes each connection it is composed of every time it is triggered by its parent block.

\[
\text{(dataflow)} \quad A, B ::= \text{data } y \text{ pre} \rightarrow x \mid \text{data } y \circ f \circ z \rightarrow x \mid \text{event } x \rightarrow y \mid A \mid B.
\]

Dataflows are structurally similar to SIGNAL programs and equally combined using synchronous composition. The interpretation \( [A]^{rt} = \{P\} \) of a dataflow (Fig. 3.12) is parameterized by the reset and trigger signals of the parent block and returns a process \( P \) (The input term \( A \) and the output term \( P \) are marked by \( \{A\} \) and \( \{P\} \) for convenience). A delayed flow data \( y \text{ pre} \rightarrow x \) initially defines \( x \) by the value \( v \). It is reset to that value every time the reset signal \( r \) occurs. Otherwise, it takes the previous value of \( y \) in time.

In Fig. 3.12, we write \( \prod_{i \leq n} P_i \) for a finite product of processes \( P_1 \mid \ldots \mid P_n \). Similarly, \( \bigvee_{i \leq n} e_i \) is a finite merge of signals \( e_1 \text{ default} \ldots e_n \) and \( \bigwedge_{i \leq n} e_i \) a finite sampling \( e_1 \text{ when} \ldots e_n \).

A functional flow data \( y \circ f \circ z \rightarrow x \) defines \( x \) by applying \( f \) to the values of \( y \) and \( z \). An event flow event \( y \rightarrow x \) connects \( y \) to define \( x \). Particular cases are the operator \( \langle y \rangle \) to convert an event \( y \) to a boolean data and the operator \( \langle y \rangle \) to convert the boolean data \( y \) to an event. We write in \( A \) and out \( A \) for the input and output signals of a dataflow \( A \).

By default, the convention of Synoptic is to synchronize the input signals of a dataflow to the parent trigger. It is however, possible to define alternative policies. One is to down-sample the input signals at the pace of the trigger. Another is to adapt or resample them at that trigger. Alternatively, adaptors could better be installed around the input and output signals of a block in order to resample them with respect to the specified frequency of the block.

\[
\begin{align*}
\text{[dataflow } & f \text{ } A]\}^{rt} = \{A\}^{rt} \cup \bigl( \prod_{x \in \text{in}(A)} x \text{ sync } t \bigr) \\
\text{[data } & y \text{ pre} \rightarrow x]\}^{rt} = \{x \text{ = (v when } r \text{) default (y pre v) } \mid (x \text{ sync } y)\} \\
\text{[data } & y \circ f \circ z \rightarrow x]\}^{rt} = \{x \text{ = } y \circ f \circ z\} \\
\text{[event } & y \rightarrow x]\}^{rt} = \{x \text{ = when } y\} \\
\text{[A } & \mid B]\}^{rt} = \{[A]\}^{rt} \cup \{B\}^{rt}\}
\end{align*}
\]

Fig. 3.12 Interpretation of dataflow connections
3.3.4 Interpretation of Actions

Actions are sequences of operations on variables that are performed during the execution of automata. The assignment \( x = yfz \) defines the value of the variable \( x \) at the next instant as the result of the application of the function \( f \) to \( y \) and \( z \). The \( \text{skip} \) action lets time elapse until the next trigger and assigns to unchanged variables the value they had at the previous instant. The conditional \( \text{if } x \text{ then } A \text{ else } B \) executes \( A \) if the current value of \( x \) is true and executes \( B \) otherwise. A sequence \( A ; B \) executes \( A \) then \( B \).

\[
\text{(action) } A, B ::= \text{skip} \mid x = yfz \mid \text{if } x \text{ then } A \text{ else } B \mid A; B .
\]

The execution of an action \( A \) starts at an occurrence of its parent trigger and shall end before the next occurrence of that event. During the execution of an action, one may also wait and synchronize with this event by issuing a \( \text{skip} \). A \( \text{skip} \) has no behavior but to signal the end of an instant: all the newly computed values of signals are flushed in memory and execution is resumed upon the next parent trigger. Action \( x! \) sends the signal \( x \) to its environment. Execution may continue within the same symbolic instant unless a second emission is performed: one shall issue a \( \text{skip} \) before that. An operation \( x = yfz \) takes the current value of \( y \) and \( z \) to define the new value of \( x \) by the product with \( f \). A conditional \( \text{if } x \text{ then } A \text{ else } B \) executes \( A \) or \( B \) depending on the current value of \( x \).

As a result, only one new value of a variable \( x \) should at most be defined within an instant delimited by a start and an end or a \( \text{skip} \). Therefore, the interpretation of an action consists of its decomposition in static single assignment form. To this end, we use an environment \( E \) to associate each variable with its definition, an expression, and a guard, that locates it (in time).

An action holds an internal state \( s \) that stores an integer \( n \) denoting the current portion of the actions that is being executed. State 0 represents the start of the program and each \( n > 0 \) labels a \( \text{skip} \) that materializes a synchronized sequence of actions.

The interpretation \( [A]^{s,m,g,E} = \langle P \rangle_{n,h,F} \) of an action \( A \) (Fig. 3.13) takes as parameters the state variable \( s \), the state \( m \) of the current section, the guard \( g \) that leads to it, and the environment \( E \). It returns a process \( P \), the state \( n \) and guard \( h \) of its continuation, and an updated environment \( F \). The set of variables defined in \( E \) is written \( \forall(E) \). We write \( \text{use}^g_{E}(x) \) for the expression that returns the definition of the variable \( x \) at the guard \( g \) and \( \text{def}^g_{E}(x) \) for storing the final values of all variables \( x \) defined in \( E \) at the guard \( g \).

\[
\text{use}^g_{E}(x) = \text{if } x \in \forall(E) \text{ then } \langle [E(x)] \rangle \text{ else } \langle (x \text{ pre } 0 \text{ when } g) \rangle, \\
\text{def}^g_{E}(E) = \prod_{x \in \forall(E)} (x = \text{use}^g_{E}(x)).
\]
3 Synoptic: A DSML for Space On-board Application Software

\[
\begin{align*}
& \text{do } A \text{ end } = \langle (P \mid s \text{ sync } t \mid r=(s=0)) \mid s \rangle \text{ where } \langle P \rangle_{n,h,F} = \big[ A \text{ end } \big] \\
& \text{end } s,n,g,E = \langle (s=0 \text{ when } g \mid \text{def}_E (E)) \rangle_{0,0} \text{, } 0 \\
& \text{skip } s,n,g,E = \langle (s=n+1 \text{ when } g \mid \text{def}_E (E)) \rangle_{n+1, (s \text{ pre } 0)=n+1} \text{, } 0 \\
& \langle x \rangle s,n,g,E = \langle (x=1 \text{ when } g) \rangle_{n,g} \\
& \langle x=yfz \rangle s,n,g,E = \langle (x=e) \rangle_{n,g,E, w[x \to e]} \text{ where } e = \langle (f(\text{use}_E^q (y)), \text{use}_E^q (z)) \text{ when } g \rangle \\
& A; B s,n,g,E = \langle P \mid Q \rangle_{n_B,B,B} \text{ where } \langle P \rangle_{n_A,A,A} = \big[ A \text{ end } \big] s,n,g,E \\
& \langle Q \rangle_{n_B,B,B} = \big[ B \text{ end } \big] s,n,g,E \\
& \text{if } x \text{ then } A \text{ else } B s,n,g,E = \langle P \mid Q \rangle_{n_B,B,B, \text{ default } E, (E_A \cup E_B)} \text{ where } \langle P \rangle_{n_A,A,A} = \big[ A \text{ end } \big] s,n,g,E \\
& \langle Q \rangle_{n_B,B,B} = \big[ B \text{ end } \big] s,n,g,E 
\end{align*}
\]

Fig. 3.13 Interpretation of timed sequential actions

Execution is started with \( s = 0 \) upon receipt of a trigger \( t \). It is also resumed from a skip at \( s = n \) with a trigger \( t \). Hence the signal \( t \) is synchronized to the state \( s \) of the store. The signal \( r \) is used to inform the parent block (an automaton) that the execution of the action has finished (it is back to its initial state 0). An end resets \( s \) to 0, stores all variables \( x \) defined in \( E \) with an equation \( x = \text{use}_E^q (x) \) and finally stops (its returned guard is 0). A skip advances \( s \) to the next label \( n+1 \) when it receives control upon the guard \( e \) and flushes the variables defined so far. It returns a new guard \( (s \text{ pre } 0) = n+1 \) to resume the actions past it. An action \( !x \) emits \( x \) when its guard \( e \) is true. A sequence \( A; B \) evaluates \( A \) to the process \( P \) and passes its state \( n_A \), guard \( g_A \), environment \( E_A \to B \). It returns \( P \mid Q \) with the state, guard and environment of \( B \). Similarly, a conditional evaluates \( A \) with the guard \( g \) when \( x \) to \( P \) and \( B \) with \( g \) when not \( x \) to \( Q \). It returns \( P \mid Q \) but with the guard \( g_A \text{ default } g_B \).

All variables \( x \in X \), defined in both \( E_A \) and \( E_B \), are merged in the environment \( F \).

In Fig. 3.13, we write \( E \cup F \) to merge the definitions in the environments \( E \) and \( F \). For all variables \( x \in \mathcal{V}(E) \cup \mathcal{V}(F) \) in the domains of \( E \) and \( F \),

\[
(E \cup F)(x) = \begin{cases} 
E(x), & x \in \mathcal{V}(E) \setminus \mathcal{V}(F), \\
F(x), & x \in \mathcal{V}(F) \setminus \mathcal{V}(E), \\
(E(x) \text{ default } F(x)), & x \in \mathcal{V}(E) \cap \mathcal{V}(F).
\end{cases}
\]

Note that an action cannot be reset from the parent clock because it is not synchronized to it. A sequence of emissions \( !x \); \( !x \) ! sends only one event along the signal \( x \) because they occur at the same (logical) time, as opposed to \( !x \); \( \text{skip} : x \text{!} \) which sends the second one during the next trigger.
Fig. 3.14 Tracing the interpretation of a timed sequential program

Example

Consider the simple sequential program of the introduction. Its static single assignment form is depicted in Fig. 3.14.

```
x = 0; if y then {x = x + 1} else {x = x - 1; skip; x = x - 1}; end
```

As in GCC, it uses a /RS-node, line 9 to merge the possible values of \( x_2 \) and \( x_4 \) of \( x \) flowing from each branch of the if.

Our interpretation implements this \( \phi \) by a default equation that merges these two values with the third, \( x_3 \), that is stored into \( x \) just before the skip line 6. The interpretation of all assignment instructions in the program follows the same pattern. Line 2, for instance, the value of \( x \) is \( x_1 \), which flows from line 1. Its assignment to the new definition of \( x \), namely \( x_2 \), is conditioned by the guard \( y \) on the path from line 1 to 2. It is conditioned by the current state of the program, which needs to be 0, from line 1 to 6 and 9 (state 1 flows from line 7 to 9, overlapping on the \( \phi \)-node). Hence the equation \( x_2 = x_1 + 1 \) when \((s = 0)\) when \( y \).

3.3.5 Interpretation of Automata

Automata schedule the execution of operations and blocks by performing timely guarded transitions. An automaton receives control from its trigger and reset signals.

---

\[1\] In the actual translation, temporary names \( x_1,...,5 \) are substituted by the expression that defines them. We kept them in the figure for a matter of clarity.
When an automaton is first triggered, or when it is reset, its starts execution from its initial state, specified as initial state \( S \). On any state \( S \) : do \( A \), it performs the action \( A \). From this state, it may perform an immediate transition to new state \( T \), written \( S \xrightarrow{on\,x} T \), if the value of the current variable \( x \) is true. It may also perform a delayed transition to \( T \), written \( S \xrightarrow{on\,x,DLE} T \), that waits the next trigger before to resume execution (in state \( T \)). If no transition condition applies, it then waits the next trigger and resumes execution in state \( S \). States and transitions are composed as \( A \mid B \). The timed execution of an automaton combines the behavior of an action or a dataflow. The execution of a delayed transition or of a stutter is controlled by an occurrence of the parent trigger signal (as for a dataflow). The execution of an immediate transition is performed without waiting for a trigger or a reset (as for an action).

\[
\text{(automaton)} \quad A, B ::= \text{state } S : \text{do } A \mid S \xrightarrow{on\,x} T \mid S \xrightarrow{on\,x,DLE} T \mid A \mid B.
\]

An automaton describes a hierarchic structure consisting of actions that are executed upon entry in a state by immediate and delayed transitions. An immediate transition occurs during the period of time allocated to a trigger. Hence, it does not synchronize to it. Conversely, a delayed transition occurs upon synchronization with the next occurrence of the parent trigger event. As a result, an automaton is partitioned in regions. Each region corresponds to the amount of calculation that can be performed within the period of a trigger, starting from a given initial state.

### Notations

We write \( \rightarrow_A \) and \( \rightarrow_A \) for the immediate and delayed transition relations of an automaton \( A \). We write \( \text{pred}_{\rightarrow_A}(S) = \{T \mid (T, x, S) \in R\} \) and \( \text{succ}_{\rightarrow_A}(S) = \{T \mid (S, x, T) \in R\} \) (resp. \( \text{pred}_{\rightarrow_A}(S) \) and \( \text{succ}_{\rightarrow_A}(S) \)) for the predecessor and successor states of the immediate (resp. delayed) transitions \( \rightarrow_A \) (resp. \( \rightarrow_A \)) from a state \( S \) in an automaton \( A \). Finally, we write \( S \) for the region of a state \( S \). It is defined by an equivalence relation.

\[
\forall S, T \in \mathcal{S}(A). ((S, x, T) \in \rightarrow_A) \iff S \equiv T.
\]

For any state \( S \) of \( A \), written \( S \in \mathcal{S}(A) \), it is required that the restriction of \( \rightarrow_A \) to the region \( S \) is acyclic. Notice that, still, a delayed transition may take place between two states of the same region.

### Interpretation

An automaton \( A \) is interpreted by a process \( \llbracket \text{automaton } x \ A \rrbracket^t \) parameterized by its parent trigger and reset signals. The interpretation of \( A \) defines a local state \( s \). It is synchronized to the parent trigger \( t \). It is set to 0, the initial state, upon receipt of
a reset signal $r$ and, otherwise, takes the previous value of $s'$, that denotes the next state. The interpretation of all states is performed concurrently.

We give all states $S_i$ of an automaton $A$ a unique integer label $i = [S_i]$ and designate with $[A]$ its number of states. $S_0$ is the initial state and, for each state of index $i$, we call $A_i$ its action $i$ and $x_{ij}$ the guard of an immediate or delayed transition from $S_i$ to $S_j$.

$$[[\text{automaton } x \ A]]^t = \langle \langle (t \ sync \ s \ s = (0 \ when \ r)) \ default (s' \ pre \ 0) \ (\prod_{S_i \in \mathcal{S}(A)} [[S_i]]^t) / s' \rangle \rangle.$$

The interpretation $[[S_i]]^t$ of all states $0 \leq i < [A]$ of an automaton (Fig. 3.15) is implemented by a series of mutually recursive equations that define the meaning of each state $S_i$ depending on the result obtained for its predecessors $S_j$ in the same region. Since a region is by definition acyclic, this system of equations has therefore a unique solution.

The interpretation of state $S_i$ starts with that of its actions $A_i$. An action $A_i$ defines a local state $s_i$ synchronized to the parent state $s = i$ of the automaton. The automaton stutters with $s' = s$ if the evaluation of the action is not finished: it is in a local state $s_i \not= 0$.

Interpreting the actions $A_i$ requires the definition of a guard $g_i$ and of an environment $E_i$. The guard $g_i$ defines when $A_i$ starts. It requires the local state to be 0 or the state $S_i$ to receive control from a predecessor $S_j$ in the same region (with the guard $x_{ji}$).

The environment $E_i$ is constructed by merging these $F_j$ returned by its immediate predecessors $S_j$. Once these parameters are defined, the interpretation of $A_i$ returns a process $P_i$ together with an exit guard $h_i$ and an environment $F_i$ holding the value of all variables it defines.

$$\forall i < [A], [[S_i]]^t = (P_i \ | \ Q_i) \ s_i \ sync \ when \ (s = i) \ s' = s_i' / s_i$$

where $\langle \langle P_i \rangle \rangle_{n, h_i, F_i} = [[A]]^{[A] \ | \ 0, g_i, E_i}$

$$Q_i = \prod_{S_j, x_{ij}, S_j} (\text{def}_{h_i \ when \ (\text{use}_{F_j}(x_{ij}))}(F_j))$$

$$E_i = \bigcup S_{\text{pred} \rightarrow A}(S_i) \ F_j$$

$$g_i = 1 \ when \ (s_i \ pre \ 0 = 0) \ default (\bigvee_{S_j, x_{ij}, S_j} \ (\text{use}_{E}(x_{ij})))$$

$$g_{ij} = h_i \ when \ (\text{use}_{F_j}(x_{ij})), \ \forall (S_i, x_{ij}, S_j) \in \rightarrow A$$

$$s_i' = (s \ when \ s_i \not= 0) \ default (\bigvee_{S_j, x_{ij}, S_j} \ (j \ when \ g_{ij}))$$

Fig. 3.15  Recursive interpretation of a mode automaton
Upon evaluation of $A_i$, delayed transition from $S_i$ are checked. This is done by the definition of a process $Q_i$ which, first, checks if the guard $x_{ij}$ of a delayed transition from $S_i$ evaluates to true with $F_i$. If so, variables defined in $F_i$ are stored with $\text{def}_{h_i}(F_i)$.

All delayed transitions from $S_i$ to $S_j$ are guarded by $h_i$ (one must have finished evaluating $i$ before moving to $j$) and a condition $g_{ij}$, defined by the value of the guard $x_{ij}$. The default condition is to stay in the current state $s$ while $s_j \neq 0$ (i.e., until mode $i$ is terminated).

Hence, the next state from $i$ is defined by the equation $s' = s_i'$. The next state equation of each state is composed with the other to form the product $\prod_{i \in [A]} s' = s'_i$ that is merged as $s' = \bigvee_{i \in [A]} s'_i$.

**Example**

Reconsider our previous sequential program, which we now represent by a mode automaton (Fig. 3.16, left). Our interpretation of automata merges, loads and stores the variables defined in states $S_1$-$5$. Thanks to a decomposition in regions, this provides an interpretation with synchronous equations that has an equivalent meaning as that of a sequential program.

One can actually check that the translation of the automaton (Fig. 3.16, right) is identical to that of the original program modulo substitution of local the local signals $x_1$-$4$ by their definition.

**3.3.6 The Polychronous Model of Computation**

The polychronous model of computation [22] defines the algebra in which the denotational semantics of Signal is expressed. In this algebra, symbolic tags $t$ or $u$ denote periods in time during which execution takes place. Time is defined by a

```
S0 : do x = 0 | S0 -> on x S1
     | S0 -> on not y S2
S1 : do x = x + 1 | S1 -> S5
S2 : do x = x - 1 | S2 -> S4
S4 : do x = x - 1 | S4 -> S2
S5 : end
```

```
x = 0 - 1 when (s = 0) when not y 
  default 0 + 1 when (s = 0) when y 
  default (x pre 0) - 1 when (s = 1) 
| s' = 1 when (s = 0) when not y 
  default 0 when (s = 0) when y 
  default 0 when (s = 1) 
| s = s' pre 0
```

Fig. 3.16 SSA interpretation of a mode automaton into dataflow equations
partial order relation $\leq$ on tags: $t \leq u$ stipulates that $t$ occurs before $u$ or at the same time. A chain is a totally ordered set of tags. It corresponds to the clock of a signal: it samples its values over a series of totally related tags. The domains for events, signals, behaviors and processes are defined as follows:

- An event is a pair consisting of a tag $t \in T$ and a value $v \in V$.
- A signal $s \in S$ is a function from a chain of tags $C \subseteq T$ to a set of values $v \in V$.
- A behavior $b \in B$ is a function from a set of names $X \subseteq V$ to signals.
- A process $p \in P$ is a set of behaviors that have the same domain.

Notations

We write $T(s)$ for the chain of tags of a signal $s$ and $\min s$ and $\max s$ for its minimal and maximal tag. We write $\mathcal{Y}(b)$ for the domain of a behavior $b$ (a set of signal names). The restriction of a behavior $b$ to $X$ is noted $b|_X$ (i.e., $\mathcal{Y}(b|_X) = X$). Its complementary $b|_X$ satisfies $b = b|_X \uplus b|_X$ (i.e., $\mathcal{Y}(b|_X) = \mathcal{Y}(b) \setminus X$). We overload $\mathcal{F}$ and $\mathcal{Y}$ to designate the tags of a behavior $b$ and the set of signal names of a process $p$. Since tags along a signal $s$ form a chain $C = T(s)$, we write $C_i$ for the $i$th instant in chain $C$ and have that $C_i \leq C_j$ iff $i \leq j$ for all $i, j \geq 0$.

Reaction

A reaction $r$ is a behavior with (at most) one time tag $t$. We write $T(r)$ for the tag of a non empty reaction $r$. A reaction $r$ is concatenable to a behavior $b$, written $b \sqcup r$, iff $r$ and $b$ have the same domain $\mathcal{Y}(b)$ and if $\max T(r) < \min T(b)$.

The concatenation of $r$ to $b$ is written $b \sqcup r$ and defined by $(b \sqcup r)(x) = b(x) \cap r(x)$ for all $x \in \mathcal{Y}(b)$.

Synchronous Structure

A behavior $c$ is a stretching of a behavior $b$, written $b \leq c$, iff $\mathcal{Y}(b) = \mathcal{Y}(c)$ and there exists a bijection $f$ on tags s.t.

\[
\forall t, u, t \leq f(t) \land (t < u \Rightarrow f(t) < f(u)), \quad \forall x \in \mathcal{Y}(b), \mathcal{F}(c(x)) = f(\mathcal{F}(b(x))) \land \forall t \in \mathcal{F}(b(x)), b(x)(t) = c(x)(f(t)).
\]

$b$ and $c$ are clock-equivalent, written $b \sim c$, iff there exists a behavior $d$ s.t. $d \leq b$ and $d \leq c$. The synchronous composition $p | q$ of two processes $p$ and $q$ is defined by combining behaviors $b \in p$ and $c \in q$ that are identical on the interface between $p$ and $q$: $I = \mathcal{Y}(p) \cap \mathcal{Y}(q)$.

\[
p | q = \{b \cup c \mid (b, c) \in p \times q \land b|_I = c|_I \land I = \mathcal{Y}(p) \cap \mathcal{Y}(q)\}.
\]
Alternatively, the synchronous structure of a process can be interpreted as an equivalence relation \( \sim \) that refines the causal tag structure of individual signals (for all \( b \in \mathcal{B} \), for all \( x \in \mathcal{V}(b) \), \( \mathcal{T}(b(x)) \) is a chain of \( \mathcal{T} \)). If we make the assumption that the only phenomenological structure is this causal structure, then the synchronous structure of a behavior \( \sim^b \) can be seen as a way to slice time across individual signals in a way that preserves causality: for all \( t, u \in \mathcal{T}(b) \), \( 1. t < u \) iff \( t \not\sim^b u \) and \( 2. t \preceq u \) iff \( t \sim u \) or \( t < v \) and \( v \sim^b u \).

### Denotation of Data-Flow Diagrams

A data-flow diagram \( fA \) is defined by the meaning of \( A \) controlled by the parent timing \( C \). A delayed flow \( \text{data } yfz \to x \) assigns \( v \) to the signal \( x \) upon reset \( t \in C' \) and the previous value of \( y \) otherwise, at time \( \text{pred}_C(x) \). An operation \( \text{data } yfz \to x \) assigns the product of the values \( u \) and \( v \) of \( y \) and \( z \) by the operation \( f \) to the signal \( x \). An event \( \text{event } y \to x \) triggers \( x \) every time \( y \) occurs. Composition \( A \mid B \) merges all timely compatible traces of \( A \) and \( B \) under the same context.

\[
\begin{align*}
\llbracket \text{dataflow } fA \rrbracket^C &= \{ b \in \llbracket A \rrbracket^C | \forall x \in \text{in}(A) C' = \mathcal{T}(b(x)) \}, \\
\llbracket \text{data } y \text{ prev } \to x \rrbracket^C &= \left\{ b \in \mathcal{B}_{x,y} \middle| \begin{array}{l}
C^y = \mathcal{T}(b(x)) = \mathcal{T}(b(y)) \cup \mathcal{T}(b(z)) \\
C^x = C^y \cup \{ \text{min}(\mathcal{T}(b(x))) \} \\
\forall t \in C^x, b(x)(t) = v \\
\forall t \in C^y \setminus C^x, b(x)(t) = b(y)(\text{pred}_C(x)) \\
\forall t \in \mathcal{T}(b(x)) = \mathcal{T}(b(y)) = \mathcal{T}(b(z)) \\
b(x)(t) = f(b(y)(t), b(z)(t)) \\
\end{array} \right\}, \\
\llbracket \text{data } yfz \to x \rrbracket^C &= \left\{ b \in \mathcal{B}_{x,y,z} \middle| \begin{array}{l}
C^x = C^y \cup \{ \text{min}(\mathcal{T}(b(x))) \} \\
C^y = \mathcal{T}(b(x)) = \mathcal{T}(b(y)) = \mathcal{T}(b(z)) \\
\forall t \in \mathcal{T}(b(x)) = \mathcal{T}(b(y)) = \mathcal{T}(b(z)) \\
b(x)(t) = f(b(y)(t), b(z)(t)) \\
\end{array} \right\}, \\
\llbracket \text{event } y \to x \rrbracket^C &= \left\{ b \in \mathcal{B}_{x,y} \middle| \begin{array}{l}
C^x = C^y \cup \{ \text{min}(\mathcal{T}(b(x))) \} \\
C^y = \mathcal{T}(b(x)) = \mathcal{T}(b(y)) = \mathcal{T}(b(z)) \\
\forall t \in \mathcal{T}(b(x)) = \mathcal{T}(b(y)) = \mathcal{T}(b(z)) \\
b(x)(t) = f(b(y)(t), b(z)(t)) \\
\end{array} \right\}, \\
\llbracket A \mid B \rrbracket^C &= \llbracket A \rrbracket^C \llbracket B \rrbracket^C.
\end{align*}
\]

### Denotation of Actions

Given its state \( b \), the execution \( c_k = \llbracket A \rrbracket_b \) of an action \( A \) returns a new state \( c \) and a status \( k \) whose value is 1 if a skip occurred and 0 otherwise. We write \( b_{1,x} = b(x)(\text{min} \mathcal{T}(b)) \) and \( b_{\text{max}} = b(x)(\text{max} \mathcal{T}(b)) \) for the first and last value of \( x \) in \( b \).

A sequence first evaluates \( A \) to \( c_k \) and then evaluates \( B \) with store \( b \cdot c \) to \( d_l \). If \( k \) is true, then a skip has occurred in \( A \), meaning that \( c \) and \( d \) belong to different instants. In this case, the concatenation of \( b \) and \( c \) is returned.

Hence, variables defined in the last reaction of \( c \) must be merged to variables defined in the first reaction of \( d \). To this end, we write \( (b \bowtie c)(x) = b(x) \bowtie c(x) \) for all \( x \in \mathcal{V}(b) \) if \( t = \text{max}(\mathcal{T}(b)) = \text{min}(\mathcal{T}(c)) \).
SPaCIFY Project

Denotation of Automata

An automaton \( x \) receives control from its trigger at the clock \( C_t \) and is reset at the clock \( C_r \). Its meaning is hence parameterized by \( C = (C_t, C_r) \). The meaning of its specifications \( \ll A \rr_b \) is parameterized by the finite trace \( b \), that represents the store, by the variable \( s \), that represents the current state of the automaton.

At the \( i \)th step of execution, given that it is in state \( j \) \( (b_i)_{\uparrow x} = j \) the automaton produces a finite behavior \( b_{i+1} = \ll S_j \rr_b \). This behavior must match the timeline of the trigger: \( \mathcal{T}(b_i) \subseteq C' \). It must to the initial state 0 is a reset occurs: \( \forall t \in C', b_i(s(t)) = 0 \).

When an automaton is in the state \( S_i \), its action \( A_i \) is evaluated to \( c \) given the store \( b \). Then, immediate or delayed transitions departing from \( S_i \) are evaluated to return the final state \( d \) of the automaton.

3.4 Middleware Aspect

In order to support the execution of code generated from Synoptic model at runtime, a middleware is embedded in the satellite. This SPaCIFY middleware not only implements common middleware services such as communication or naming, but also offers domain-specific services of satellite flight control software. There are then
two difficulties that must be addressed. First, one has to take into account the scarcity of resources in a satellite. The large variety of satellite platforms from one project to another being the second.

To take into account the scarceness of resources in a satellite, this middleware is tailored to the domain and adapted to each specific project. This notion of generative middleware is inherited from the ASSERT project which has studied proof-based engineering of real-time applications but is here specialised to the area of satellite software. ASSERT defines a so-called virtual machine, which denotes a RTOS kernel along with a middleware and provides a language-neutral semantics of the Ravenscar profile [14]. It relies on PolyORB-HI [25], a high integrity version of PolyORB refining the broker design pattern [15], which fosters the reuse of large chunks of code when implementing multiple middleware personalities. Satisfying restrictions from the Ravenscar profile, PolyORB-HI can suitably be used as a run-time support for applications built with AADL code generators. In our context, the support of a precise adaptation to the need of the middleware is obtained thanks to its generation based on the requirements expressed in the Synoptic models (mainly through the interaction contracts attached to external variables).

Some of the services of the middleware cannot be reused and must be reimplemented for each satellite. For example, the AOCS\(^3\) implements control laws that are specific to the expected flight and to the mission. Providing a framework, the middleware will help capitalizing on the experience of domain expert, giving a general architecture for the AOCS. The models of services can belong to either the middleware or the application. In the later case, we use the same process as for the rest of the application, including the Synoptic language. Furthermore as services will have a well defined interface (i.e., an API) other services or applications using AOCS are not coupled to any particular implementation.

This section starts by presenting the architecture of the SPaCIFY middleware. Then, the middleware kernel and the communication between applications generated from Synoptic models and their hosting middleware is explained. Lastly, the reconfiguration service that has been one of the main focus during the project is described.

### 3.4.1 Architecture of the Execution Platform

Figure 3.17 depicts the overall architecture of the SPaCIFY middleware. Following previous work on middlewares for real-time embedded systems [8, 32], the SPaCIFY middleware has a microkernel-like or service-based architecture. That way, high flexibility allows to embed only the services that are required, depending on the

---

\(^2\) This diversity is due to the high specialisation of platforms for a specific mission and the broad range of missions.

\(^3\) Attitude and Orbit Control System.
synchronous islands

asynchronous
environment

Management of external variables

Fig. 3.17 Architecture of the middleware

The RTOS kernel offers the usual basic services such as task management and synchronization primitives. The core middleware is built on top of this RTOS and provides core services of composition and communication. They are not intended to be used by application-level software. They rather provide means to structure the middleware itself in smaller composable entities, the services. Upon these abstractions are built general purpose services and domain specific services. These services are to be used (indirectly) by the application software (here the various synchronous islands). They can be organized in two categories as follows:

- The first layer is composed of general purpose services that may be found in usual middleware. Among them, the naming service implements a namespace that would be suitable for distributed systems. The persistency service provides a persistent storage for keeping data across system reboots. The redundancy service helps increasing system reliability thanks to transparent replication management. The reconfiguration service, further described in a dedicated subsection below (Sect. 3.4.3), adds flexibility to the system as it allows to modify the software at runtime. The task and event service contributes real-time dispatching of processors according to the underlying RTOS scheduler. It provides skeletons for

- Using application model to specify reconfiguration
- Proposing a contract approach to the specification of relation between applications and the middleware
- Providing on demand generation of the needed subset of the middleware for a given satellite
various kinds of tasks, including periodic, sporadic and aperiodic event-triggered
tasks, possibly implementing sporadic servers or similar techniques [26, 33].

- The second layer contains domain-specific services to capture the expertise in the
  area of satellite flight control software. These services are often built following
  industry standards such as PUS [17]. The TM/TC service implements the well
  established telemetry/telecommand link with ground stations or other satellites.
The AOCS (attitude and orbit control system) controls actuators in order to en-
  sure the proper positioning of the satellite. As discussed earlier, services may be
  implemented by synchronous islands.

To support the execution the platform use hardware resources provided by the
  satellite. As the hardware platform changes from one satellite to another, it must
  be abstracted even for the middleware services. Only the implementation of spe-
  cific drivers must be done for each hardware architecture. During the software
development lifecycle, the appropriate corresponding service implementations are
  configured and adapted to the provided hardware.

As already exposed, one particularity of the SPaCIFY approach is the use of
  the synchronous paradigm. To support the use of the middleware services within
  this approach, we propose to use an abstraction, the external variable as shown in
  Sect. 3.2.2.2. Such a variable abstracts the interaction between synchronous islands
  or between a synchronous island and the middleware relaxing the requirements of
  synchrony. In the model, when the software architect wants to use a middleware
  service, he provides a contract describing its requirements on the corresponding
  set of external variables and the middleware is in charge to meet these require-
  ments. Clearly, this mediation layer in charge of the external variables management
  is specific to each satellite. Hence the contractual approach drives the generation of
  the proper management code. This layer is made of backends that capture all the
  asynchronous concerns such as accessing to a device or any aperiodic task, hence
  implementing asynchronous communications of the GALS approach. The middle-
  ware is in charge of the orchestration of the exchange between external variables,
  their managing backends and the services while ensuring the respect of quality of
  service constraints (such temporal one) specified in their contracts.

### 3.4.2 The Middleware Kernel and External Variables

As stated above, the middleware is built around a RTOS providing tasks and
  synchronisation. As the RTOS cannot be fixed due to industrial constraints, the
  middleware kernel must provide a common abstraction. It therefore embeds its own
  notion of task and for specific RTOS an adaptor must be provided. The implemen-
  tation of such adaptors has been left for future until now. Notice that the use of
  this common abstraction forbids the use of specific and sophisticated services of-
  fered by some RTOS. The approach here is more to adapt the services offered by
  the middleware to the business needs rather using low level and high performance
  services.
The task is the unit of execution. Each task can contain Synoptic components according to the specification of the dynamic architecture of the application. Tasks have temporal features (processor provisioning, deadline, activation period) inherited from the Synoptic model. The middleware kernel is in charge of their execution and their monitoring. It is also in charge of provisioning resources for the aperiodic and sporadic tasks.

Communication inside a task results from the compilation of the synchronous specification of the various components it must support. All communication outside a task must go through external variables limiting interaction to only one abstraction. External variables are decomposed into two sub abstractions:

- The frontend identified in the Synoptic model and that constitutes the interaction point. It appears as a usual signal in the synchronous model and may be used as input or output. The way it is provided or consumed is abstracted in a contract that specify the requirements on the signal (such as its timing constraints). An external variable is said to be asynchronous because no clock constraint is introduced between the producer of the signal and its consumer. In the code generated access to such variables are compiled into getter and setter function implemented by the middleware. The contract must include usage requirements specifying the way the signal is used by the task (either an input or an output). They may also embed requirements on the freshness of the value or on event notifying value change.

- The backend specified using stereotypes in the contract configure the middleware behavior for non-synchronous concerns. For example, persistency contract specify that the external variable must be saved in a persistent memory. Acquisition contracts can be used by a task to specify which data it must get before its execution. Such backends are collected and global acquisition plan are built and executed by the middleware.

As the middleware supports the reconfiguration of applications at runtime, tasks can be dynamically created. The dynamic modification of the features is also provided by the middleware. The middleware will have to ensure that these constraints are respected. Beware that any modification must be made on the model and validated by the SPaCIFY tools to only introduce viable constraints. Each task has a miss handler defined. This defensive feature makes the middleware execute corrective actions and report an error whenever the task does not respect its deadline.

### 3.4.3 Reconfiguration Service

Among, domain specific services offered by the middleware, the reconfiguration service is the one that had the most impact on the middleware conception.

The design of the reconfiguration service embedded in the SPaCIFY middleware is driven by the specific requirements of satellite onboard software. Reconfiguration is typically controlled from ground stations via telemetry and telecommands.
Human operators not only design reconfigurations, they also decide the right time at which reconfigurations should occur, for instance while the mission is idle. Due to resource shortage in satellites, the reconfiguration service must be memory saving in the choice of embedded metadata. Last, in families of satellites, software versions tend to diverge as workarounds for hardware damages are installed. Nevertheless, some reconfigurations should still apply well to a whole family despite the possible differences between the deployed software.

In order to tackle these challenges, the reconfiguration service rely on the structural model (Synoptic models) of the OBSW enriched by the current state of the satellite as described in Fig. 3.18. Using the structure of the software allows to abstract the low-level implementation details when performing reconfigurations. While designing reconfigurations, operators can work on models of the software, close to those used at development time, and specify so called abstract reconfiguration plan like: \textit{change the flow between blocks A and B by a flow between A and C}. An abstract reconfiguration plan use high level elements and operations which may increase the CPU consumption of reconfigurations compared to low level patches. Typical operations such as pattern matching of components make the design of reconfiguration easier, but they are compute intensive. Instead of embedding the implementation of those operations into the satellite middleware, patterns may be matched offline, at the ground station, thanks to the knowledge of the flying software. We therefore define a hierarchy of reconfiguration languages, ranging from high-level constructs presented to the reconfiguration designer to low-level instructions implemented in the satellite runtime. Reconfigurations are compiled in a so called concrete reconfiguration plan before being sent to the satellite. For instance, the abstract plan \textit{upgrade A} may be compiled to \textit{stop A and B; unbind A and B; patch the implementation of A; rebind A and B; restart A and B}. This compilation process uses proposed techniques to reduce the size of the patch sent to the satellite and the time to apply it [35]. Another interest of this compilation scheme is to enable the use of the same abstract reconfiguration plan to a whole family of satellites. While traditional patch based approaches make it hard to apply a single reconfiguration to a family, each concrete plan may be adapted to the precise state of each of the family member.
Applying the previously model driven approach to reconfigurations of satellite software raise a number of other issues that are described in [10]. Work remains to be conducted to get the complete reconfiguration tool chain available. The biggest challenge being the specification of reconfiguration plan. Indeed, considering reconfiguration at the level of the structural model implies to include the reconfigurability concern in the metamodel of Synoptic. If reconfigurability is designed as an abstract framework (independent of the modeling language), higher modularity is achieved when deciding the elements that are reconfigurable. First experiments made on Fractal component model [11] allow us to claim that focusing on the points of interest in application models, metadata for reconfiguration are more efficient. Indeed, separating reconfigurability from Synoptic allows to download metadata on demand or to drop them at runtime, depending on requested reconfigurations.

Lastly, applying reconfiguration usually require that the software reach a quiescent state [28]. Basically, the idea consists in ensuring that the pieces of code to update are not active nor will get activated during their update. For reactive and periodic components as found in the OBSW, such states where reconfiguration can be applied may not be reached. We have proposed another direction [12]. We consider that active code can be updated consistently. Actually, doing so runs into low-level technical issues such as adjusting instruction pointers, and reshaping and relocating stack frames. Building on previous work on control operators and continuation, we have proposed to deal with the low level difficulties using the notion of continuation and operators to manipulates continuations. This approach do not make updating easier but gives the opportunity to relax the constraints on update timing and allow updates without being anticipated.

3.5 Conclusion

Outlook

The SPaCIFY ANR exploratory project proposes a development process and associated tools for hard real time embedded space applications. The main originality of the project is to combine model driven engineering, formal methods and synchronous paradigms in a single homogeneous design process.

The domain specific language Synoptic has been defined in collaboration with industrial end-users of the project combining functional models derived from Simulink and architectural models derived from AADL. Synoptic provides several views of the system under design: software architecture, hardware architecture, dynamic architecture and mappings between them. Synoptic is especially well adapted for control and command algorithm design.

The GALS paradigm adopted by the project is also a key point in the promoted approach. Synoptic language allows to model synchronous islands and to specify how these islands exchange asynchronous information by using the services of a dedicated middleware.
In particular, the SPaCIFY project proposed a contract approach to the specification of the relations between applications and the middleware. This middleware does not only implement common services such as communication or naming, but also offers domain-specific services for satellite flight control software. Reconfiguration is one of these domain-specific services. The SPaCIFY project studied this service with a particular focus on the use of application model to specify reconfiguration.

The Synoptic Environment

The SPaCIFY design process has been equipped with an Eclipse-based modeling workbench. To ensure the long-term availability of the tools, the Synoptic environment rely on open-source technologies: it guarantees the durability and the adaptability of tools for space projects which can last more than 15 years. We hope this openness will also facilitate adaptations to other industries requirements.

The development of the Eclipse-based modeling workbench started with the definition of the Ecore meta-model of the Synoptic language. The definition of this meta-model has relied on the experience gained during the GeneAuto project. This definition is the result of a collaborative and iterative process. In a first step, a concrete syntax relying on the meta-model has been defined using academic tools such as TCS (Textual Concrete Syntax). This textual syntax was used to validate the usability of the language through a pilot case study. These models have helped to improve the Synoptic language and to adapt it to industrial know-how. Once the language was stabilized, a graphical user editor was designed. A set of structural and typing constraints have been formalized, encoded in OCL (Object Constraint Language), and integrated into the environment.

In parallel of these activities, the semantics of the language was defined. The formal semantics of the Synoptic language relies on the polychronous paradigm. This semantics was used for the definition of the transformation of Synoptic models towards SME models following a MDE approach. This transformation allows to use the Polychrony platform for verification, semantics model transformation hints for the end user, such as splitting of software as several synchronous islands and simulation code generation purposes.

The Synoptic environment is being used to develop case studies of industrial size.

Future Investigations

The Synoptic environment is based on model transformation. Thus, verifying this transformations is a key point. It has been addressed in the Geneauto project to certify sequential code generation from a Stateflow/Simulink based language. This work must be extended to take into account features of the execution platform such as timers, preemption-based schedulers, multi-threading, multi-processors, ... Work is in progress on a subset of the Synoptic language.
The Synoptic environment provides a toolset supporting a development process. Experience acquired during the SPaCiFY project with industrial partners has enlightened two different processes and thus the need to parameterize the platform by the process. A SPEM-based specification of the development process could be used as input of a generic platform so that it could be configured to match end user current and future development method.

The Synoptic environment offers a limited support to refinement-based development process. This support could be extended and coupled with versioning to allow refinement checking between any couples of models or submodels. It means a support for defining and partially automatically generating gluing invariants between models. Then proof obligations could be generated.

References

3 Synoptic: A DSM for Space On-board Application Software


17. ESA. European space agency. ground systems and operations – telemetry and telecommand packet utilization (ECSS-E-70), January 2003.


