Fernández, J., Parra, P., Sánchez-Prieto, S., Polo, O. & Bernat, G. 2015, "Automatic verification of timing constraints for safety critical space systems", in Proceedings DASIA 2015, DAta Systems In Aerospace, 19-21 May, 2015, Barcelona, Spain. Edited by L. Ouwehand, ESA-SP, vol. 732, id 63

Available at http://adsabs.harvard.edu/abs/2015ESASP.732E..63F

AUTOMATIC VERIFICATION OF TIMING CONSTRAINTS FOR SAFETY CRITICAL SPACE SYSTEMS

Javier Fernandez[1], Pablo Parra[1], Sebastian Sanchez Prieto[1], Oscar Polo[1], and Guillem Bernat[2]

[1]*Space Research Group, Universidad de Alcala, Madrid, Spain*
[2]*Rapita Systems Ltd, York, England, UK*

ABSTRACT

In this paper is presented an automatic process of verification. We focus in the verification of scheduling analysis parameter. This proposal is part of process based on Model Driven Engineering to automate a Verification and Validation process of the software on board of satellites. This process is implemented in a software control unit of the energy particle detector which is payload of Solar Orbiter mission. From the design model is generated a scheduling analysis model and its verification model. The verification as defined as constraints in way of Finite Timed Automatas. When the system is deployed on target the verification evidence is extracted as instrumented points. The constraints are fed with the evidence, if any of the constraints is not satisfied for the on target evidence the scheduling analysis is not valid.

Key words: V&V; Automatic Verification; Embedded Software; Energy Particle Detector; MDE ; CBSE.

1. INTRODUCTION

The verification of embedded software in space applications is one of the critical tasks in the development process. The verification activities are specified in detail in several standards, in particular ECCS-E-ST-E40 (1) for the development of space applications. The introduction of model-based software engineering approaches (MDE) (2) and component-based software development (CBSE) (3) introduces new challenges in the verification of the overall system. The current effort required to provide enough evidence is costly and time-intensive. The increase in complexity, functionality and overall size of software in space makes the verification aspects even more critical. In this context, approaches that provide automatic verification of evidence as part of the software development process result in a higher level of assurance, therefore addressing the growth in complexity with a lower overall effort and cost.

In the context of this paper, we draw a very clear distinction between Validation and Verification (V&V)(4). In the context of the "V" development process, verification is the process of reviewing and ensuring that each step of the development process is consistent and complete in itself. This process can be illustrated by asking the reader a simple question (are we building the product right?), whereas validation is the process of testing that the process meets the requirements ("are we building the right product?"). In the context of this paper, verification is the process of producing evidence of reviewing the evidences that assure the software conforms with requirements, design constraints and analysis hypotheses.

A motivational example is the verification objective related to schedulability analysis. This includes specifying the scheduling type (sequential or multi-task), the scheduling model (e.g. cyclical or preemptive, fixed or dynamic priority) and the scheduling algorithm (e.g. fixed-priority preemptive). In multi-task systems, the analysis also takes into account how resources are shared, and what parameters of the scheduling, including periods, deadlines and worst-case execution times, are used. From all this information, and by applying schedulability analysis, we can calculate the worst-case response time and determine if deadlines are always met (5). The schedulability analysis is correct as long as the input parameters are correct and guaranteeing that the implementation follows the scheduling model [2]. The key weakness of this approach is to provide correct parameters to the scheduling model. If the parameters are not correct, the obtained results are unreliable. This could happen if:

- The tasks are executed with wrong priority or periods.
- The Real Time Operating System (RTOS) implements priority levels different from the programmers

This work were supported by the MINECO under the project ESP2013-48346-C2-2-R

[2]The assurance that the scheduling tool and the implementation of the schedulability analysis is correct belongs to the realm of tool qualification and is not addressed in this paper

assumption[3]

- The estimated WCET is not correct.
- The RTOS implements the scheduling incorrectly.
- There exists a priority inversion not addressed in the analysis [4]
- The hardware is configured incorrectly leading to a time base on the RTOS that is not real-time.
- Interrupts are assumed to be disabled, but an improper configuration of interrupt handlers still allows interrupts to be raised.
- There are operations on shared resources that have not been modeled by the schedulability analysis. A typical example of these operations is hidden mutexes. These mutexes are inside either library modules or the RTOS.

In any of these scenarios the behavior of the system on target is different from the one obtained from the models and therefore any schedulability analysis is wrong. The problem is therefore how to provide evidence that show whether the assumptions under which the scheduling model is performed are or are not preserved in the implementation. A manual process for determining all this evidence is not only extremely effort-intensive, but error-prone and impractical. Some of these scenarios may lead to a non-functioning system and, as such, would be relatively easy to resolve. The problem arises from those rare events that do not manifest themselves until the integration tests. Consequently, they can be by nature extremely difficult to find and replicate.

The hypothesis of this paper is that there exists a process for automatic verification of a set of assumptions and properties from high level models. This process can be fully supported by tools that are able to perform an automatic verification of the properties. This results in greater confidence in the correctness of the analysis and the implementation. At the same time, it enables an early identification of the violations of these assumptions.

The underlying principle that has motivated this work is the statement"you shall trust no one". By providing a systematic and automatic process for verifying each of the transformations from evidence of the execution of the system in the final target it is possible to reduce, if not remove, the need for assumptions that a transformation is correct, or at least provide evidence that a set of hypotheses or assumptions are preserved. We understand that it is not possible to provide perfect bug-free systems, but the quality of current development practices can be raised at a lower cost by adopting a systematic approach in order to verify the expected behavior of the final system.

[3] large numbers are assigned to lower priorities instead of low numbers to low priorities or vice versa.

[4] A notorious example of this situation is the Mars Pathfinder priority inversion.

The process is based on the specification of the assumptions at the different stages of the design and analysis process as constraints described as finite timed automata. The alphabet of these constraints are events in the execution of the final system. An instrumentation step adds lightweight placeholders in the source code to observe these events. This can be performed by a transparent automatic process. As a consequence, the execution of the system on target (or representative hardware) results in a timing trace of the execution that can then be checked against the list of constraints. A tool can then produce a report that shows that all constraints have always been satisfied (positive constraints) or that a constraint has never been achieved (negative constraints). The end result is a report that produces the necessary evidence (qualifiable) that can be used as part of the certification process.

2. THE PROCESS TOOLSET

The EDROOM (6) tool is based on a Component-Based Software Engineering model (CBSE). EDROOM is similar to tools such as: SOFA and ObjectTime (7; 8) (in fact EDROOM is inspired by latter, but more focused on embedded systems). It defines the communication among components in terms of protocols, and the components instantiate ports associated to protocols as communication interfaces. The behaviour description of the components is defined as ROOMcharts. ROOMcharts are based on Harel state charts (9), and they are semantically equivalent to UML2 statecharts (10).

The EDROOM tool generates code application from the EDROOM model description. The code generated is supported by a component runtime, called EDROOM Service Library, that is based on a two-layer architecture. The top layer, which is independent of the platform, provides a service interface for code application (task and task priority management, subscription to message services, mutexes, etc.); the bottom layer provides the interface with the RTOS. The current version of the EDROOM runtime supports RTEMS, CMX, RTAI and Linux.

Model-driven engineering (MDE) techniques support the definition of software processes. It defines the process as models and model transformations. This implies a high degree of model cohesion and continuity of the models. In this paper, this paradigm is supported by the MICOBS framework (11). The MICOBS targets the requirements of the development process of embedded real-time systems, as it is the on-board satellite software. Central to the framework is a platform-aware approach for model annotation. It enables the introduction of extra-functional properties in the model specific to each potential deployment platform and configuration parameters. In addition, it enables the definition of transactional models that make it easy to implement model transformations. The transactional analysis is the main goal of this work. As part of the automatic transformation, a set of constraints that define the transformation are also generated. These con-

straints can be automatically verified by evidence of the execution of the final systems execution on the target. The transactional model is widely explained in section 3.

RapiCheck is a tool developed by Rapita Systems Ltd. as part of their toolset for software verification: Rapita Verification Suite (RVS)[5]. Constraints are specified in the RapiCheck language and provided as an input on the instrumentation process. This process determines the alphabet of the constraints as events in the code execution (for example, entry points in a function call, end of a function, etc.) and automatically adds instrumentation points to the code which will generate a trace of the execution when the system runs. When the system is compiled and run on the target, a data collection mechanism captures the trace of the execution (only the events that need to be observed) and sends the trace to a host. The final stage consists of processing the trace against the constraints and providing a report that shows which constraints are satisfied or failed and in which cases they have been satisfied in the trace. The constraint language of RapiCheck is based on a timed automata with arbitrary guards.

Figure 1 shows the global process that automates the generation of the scheduling analysis and the RapiCheck which verifies a set of evidence in the target. The structures of transactional models of analysis are generated from the EDROOM models. In unitary tests these models of analysis are annotated along with the execution time. This is done thanks to the RVS 3.0 tool and the use of a logic analyzer. From transactional models of analysis, both the scheduling analysis model as well as the corresponding models of verification, can be generated.

3. TRANSACTIONAL ANALYSIS MODELS

The transactional analysis models are composed for a set of models. Each of these models describes, from an orthogonal perspective, the real-time analysis based on the CBSE paradigm. The Transactional Component Description AOM is responsible for describing the extra-functional properties related to the behaviour of the components. The Transactional Platform Description model is responsible for defining the extra-functional properties of the platform (run-time and hardware). The Flat System Deployment AOM is responsible for describing the system depending on the component composition. Finally, the Transactional Real-Time Requirement model describes the events and the time constraints. The following paragraphs will provide a detailed description of each one of the afore mentioned models.

The Transactional Component Description model describes the reactive behaviour of the components in terms of the messages that can be received. Each action is specified using an artifact such as `TSAMMessageHandler`. This elements is composed of the definition of a tuple (port, message) and by a sequence of execution items called `TSAMBasicHandler`. The tuple (port, message) indicates when the handler `TSAMMessageHandler` is triggered, and the `TSAMBasicHandler` elements specify the sequence of steps that comprises the message response.

There are two types of `TSAMMesssageHandler`, which can be categorized according to the message that triggers them. The definition of `TSAMAsynchMsgHandler` is shown in Figure 2 and it models the response to an asynchronous message reception. A `TSAMAsynchMsgHandler` is triggered by receiving an asynchronous message from another component, or from the run-time system as a result of an event. The other type is `TSAMSynchMsgHandler` which models the response to a synchronous messages invoked from other component.

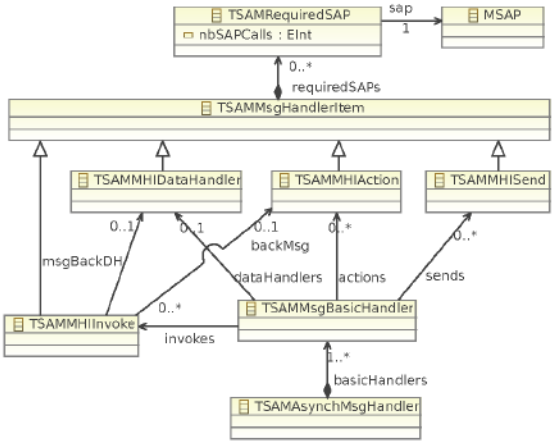

Figure 2. Transactional analysis model of asynchronous message handlers.

As it was noted before, each `TSAMMessageHandler` defines the sequence of `TSAMBasicHandler` that comprises the message response. Each `TSAMBasicHandler`, is composed by different items that are executed sequentially. These items are called `TSAMMHIItems`. There are four types of `TSAMMHItems`, as specified below:

`TSAMMHiAction` : Defines an action whose effect is restricted to the scope of the `TSAMBasicHandler`. This action does not involve the sending of any message, nor the handling of data attached to the received message.

`TSAMMessageSend` : Sends an asynchronous message to another component. The port and the message to be sent are the parameters.

`TSAMMHiInvoke` : Sends a synchronous message to a

[5]http://www.rapitasystems.com/

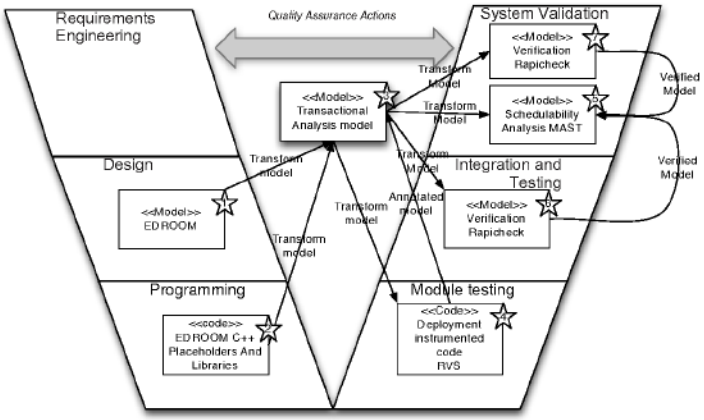

Figure 1. The whole Validation and verification process.

component. The sending port and the message are the parameters.

`TSAMMHiReply` : Defines the answer to a synchronous message (`TSAMMHiInvoke`). The response message must be specified.

Two components types have been defined, task components and shared resource components. These components have restrictions related to the `TSAMMessageHandler` type they can contain and their relationship with the systems execution threads. Proactive components can contain the `TSAMAsychMessageHandler` and `TSAMSynchronous`. Shared resource components can only contain `TSAMSynchronous` Aside from this, task components define their own task, executing the `TSAMMHItems` of each `TSAMAsychMessageHandler` on its task. Reactive components have no associated task. This constraint on items by `TSAMMessageHandler` and the definition of components is related to the fact that the model is free from effects related with deadlock phenomena.

Each `TSAMHIItem` is annotated with values for the worst-case execution time (WCET). As mentioned previously, the WCET can be characterized during the unitary tests.

Each component can define more than one `TSAMMessageHandler` per tuple (port, message). The one that is selected for execution depends on the components internal state, and the real-time situations at system level. Each real-time situation determines which `TSAMMessageHandlers` can happen in it. This fact enable the composition of the end-to-end flow of the scheduling analysis in order to choose only those `TSAMMessageHandlers` that correspond to the real-time situation to be analyzed. Consequently as previously mentioned, our model is restricted to a single `TSAMMessageHandler` by the following elements: port, message and real-time situation.

Once the components are defined, they are instantiated in order to build the real-time system. The flat deployment model describes the instances of the components and the topology of the communication. The communication topology resolves the connections between the component ports.

The extra-functional properties of the platform are defined in a platform model. These properties include: context switch time, timers and alarms, interrupt handler routines and hardware jitter. The users must note each of the properties in the scheduling analysis.

Finally, a real-time requirement model compiles the system level information that is required to complete the schedulability analysis. This model defines the event activation patterns and the event deadlines. Three activation patterns can be defined, periodic, sporadic and bursty. The parameter for periodic events is the frequency. The parameter for sporadic events is the minimum arrival time between two successive activations. The pattern for the bursty events has two parameters, the minimum-interarrival time and the maximum number of elements per burst. Each event defines the tuple (component, port, message) associated to its trigger. The deadlines specify the maximum time that can elapse from the activation of an event until a specific `TSAMMhItem`.

A set of real-time requirements models can be specified with the objective of specifying different real-time situations. Each one will contain different activation events and time constraints. The name of each model will be used as a tag to record which `TSAMMessageHandlers` are activated for a given real-time situation.

4. CONSTRAINTS

A set of constraints profiles have been defined with the objective of verifying the evidence in the system tar-

get. Evidence verify on one hand, the behaviour and the WCET as annotations in the transactional models, and on the other hand, that the transformations from the design model to the analysis model are correct. During the integration test a combination of profiles to be verified may be chosen. The steps that have to be followed are: 1) enable the profiles to be verified, 2) automatically generate and implement the Timed Finite Automata (TFA), 3) run the integration test and 4) extract the target trace using a data logger or logic analyzer and verify the restrictions with the profiles that were generated before using a Rapicheck tool.

The defined profiles are: 1) End-to-End flow profile, responsible for checking that the `TSAMMessageHandler` sequence triggered by an event is defined in the transactional model, 2) `TSAMMHIItems` WCET profile that checks if the WCET associated with the `TSAMMHIItems` is listed in the transactional models, 3) Platform jitter and WCET runtime primitives profile that verifies if the WCET and jitters associated with the platform and run-time are correct, 4) Real-time requirements deadlines profile that verifies if the deadlines are defined. It must check that the deadline is not greater than the elapsed time from the activation of the event to the `TSAMMessageHandlerItem` execution deadline, and 5), Event pattern trigger profile that verifies if the parameters associated with the activation patterns are specified, if the WCET and the jitter are correct and if the data obtained from the tests do not exceed the WCET. In this paper we will focus on the End-to-End and Real-time situation profile.

With the aim of demultiplexing different events a point identifier has been defined. The reason for this implementation is that the same `TSAMMHItem` can be executed for different events. If separate checking is desired for two execution sequences associated with two events that run the same `TSAMMHItem`, the trace must be demultiplexed. It consists of the following elements: 1) Event-seed, which identifies a separate event and it is randomly generated. 2) Event-identifier, which identifies the event. 3) The identifier of the specific point to be checked.

4.1. End-to-End profile

For each event, a verification element is generated in the form of a Rapicheck FTA. The transactional models required to generate this profile are the real-time requirement model, the transactional component model, and the flat system deployment model. The `TSAMEvents` are extracted from the real-time requirement model. The sequence of actions that makes the system react can be extracted from each `TSAMEvent`. The first element in this sequence is the `TSAMMessageHandler` that is associated with the (port, message) tuple, and with the component that initially handles the event. All the `TSAMMessageHandlers` are contained in the transactional component model along with each `TSAMHItems`. Each `TSAMHItems` is transformed into a Rapicheck state. The precedence of the states will depend on the order defined in the `TSAMMessageHandler` and how the `TSAMMHISend` or `TSAMMHIInvoke` are resolved. In order to resolve that `TSAMMessageHandler` reacts to the `TSAMMHISend` or `TSAMMHIInvoke`, the connections between ports must be known. These connections are defined in the Flat System deployment model.

The `TSAMMHIInvoke` and `TSAMMHISend` types generate new dependencies on other components that have to be resolved because these operations model the communication between components and generate new subsequences of states in the resulting Rapicheck FTA.

The thread that executes a `TSAMMHInvoke` waits for the reception of a `TSAMMHIReply` from the reactive component that handles the synchronous message. This behaviour is solved by first seeking the `TSAMSynchronous` that satisfies the synchronous call and then by generating two states in Rapicheck, one corresponding to `TSAMMHInvoke` and then another corresponding to `TSAMMHIReply`.

The `TSAMMHISend` are more complex to solve due to the fact that they are executed in the context of another task. Execution thus depends on the task priority and systems architecture. In mono-processor systems such as ERC-32, LEON2, etc., the `TSAMMessagehandlers` that handle the message are executed sequentially so the issue is solving the order of precedence. This will depend on the priority of each component. The algorithm is a tree search, where each `TSAMMessageHandler` is a node, and the arcs contain the priority associated with the component where the `TSAMMessageHandler` first visits the highest priority nodes. The visit order of the `TSAMMessageHandler` is the sequence in which the `TSAMMhItems` must be included.

A set of `TSAMMessagehandler` can be associated with the same (port, message) tuple. This structure is transformed in Rapicheck, generating different arcs, one for each `TSAMMessagehandler`. The `TSAMMhItems` of each one is solved separately using the transformation rules described above. It should be noted that for each branch a list of `TSAMMHISend` must be resolved and the order of precedence will be altered. If the `TSAMMhItems` sequence is the same for two branches, they are simplified. This aspect will be clarified in 6.

4.2. Real-time situation profile

This profile allows the real-time situations defined in the rt-requirement model to verified. We saw in section 2 that each `TSAMMessageHandler` in the transactional component model can have a relationship 1 to n with real-time situations. This means that the elements of these `TSAMMessageHandlers`, the `TSAMMhItems`, are only executed when a specific real-time situation is activated in the system. A real-time situation consists of

an identifier and a modulation point. A modulation point consists of two parameters, the real-time input status and the TSAMMessageHandler indicating the activation of the real-time situation. When a change in a real-time situation is produced, the new TSAMEvents can only execute the TSAMMessageHandler corresponding to this real-time situation.

A Rapicheck FTA is generated in order to check that the order of real-time situation activation corresponds to the defined modulation points. This allows for the generation of a state for each real-time situation. The arcs of this FTA are the modulation points, and to transit from one state to another is through the execution of the TSAMMessageHandlers which are selected as modulation point.

Once a real-time situation is activated, it must be verified that the TSAMMessageHandlers are related to that real-time situation. In order to verify this aspect, one Rapicheck FTA per real-time situation is generated and one per TSAMEvent. This FTA is generated in the same way as those for the End-to-End profile. but they will only contain the TSAMMessageHandlers related to the real-time situation to be verified. The elements of the trace generated by the integration test may contain more than one real-time situation. This implies that the trace must be divided into as many traces as there are defined real-time situations, in order to verify each of the FTAs relating to each TSAMEvent and real-time situation. The division is easy to perform since the modulation points become trace elements. It should be noted that the TSAMEvent for which their execution was not finalized should be included in this trace. These elements are easy to identify as the event-seed identifiers after the modulation point are still those belonging to the previous real-time situation. Once each FTA is fed, it is possible to check if the evidence matches the model.

5. INSTRUMENTATION AND OVERHEADS

This proposed verification is based on evidences that are obtained by code instrumentation. This instrumentation generates overheads in the execution times so this has to be taken into account. It would be interesting to generate the smallest number of elements to be instrumented in order to verify all the previously cited profiles. The instrumentation policy is performed for the profiles that have been enabled, so that the instrumentation code is the smallest possible in order to allow all the profiles to be observed. In fact, if the End-to-End profile is active only the TSAMMessageHandlerItem, WCET profile will generate new instrumentation points, meaning that additional overhead is avoided. It should be noted that the overhead will depend on the number of instrumentation points and on its execution periodicity. For example, elements to be verified relative to jitters or operating system context switches generate a high overhead in the system.

One of the most effective ways for reducing the overhead is through the specification of levels of detail for each of the profiles. A total of three levels of detail have been defined. The most detailed level takes into account the TSAMMHItem into the account, the intermediate level takes into account the theTSAMBasicHandler into account and, finally, the lowest level takes into account the TSAMMessageHandler into account. This means that only the first and last elements in the level of detail along with the connections points are instrumented with other TSAMMessagehandlers such as TSAMMHISend. For example, a TSAMBasicHandler level of detail that contains a TSAMMHISend is instrumented in the following way: the start of the first TSAMMHItem of the TSAMBasicHandler, the first TSAMMHItem of the start of the TSAMMessagehandlers that is resolved with the TSAMMHISend and the end of the last TSAMMHItem; this way the WCET will correspond to the sum of the TSAMMHItems contained in the sequence.

6. CASE STUDY

The aim of this study is to verify the flight software for the Instrument Control Unit (ICUSW) of the Energetic Particle Detector (EPD) instrument on board of the Solar Orbiter satellite . The ICUSW is the interface between the EPD sensors and the missions spacecraft. The onboard computer interface provides management of the telecommands that change the instruments behaviour. The sensors interface manages the telemetry and housekeeping.

Figure 3 shows a diagram of the ICUSWs EDROOM model. Four tasks have been defined to meet the functional requirements. The specific roles of each one are the following:

EPDManager captures the telecommands sent from the spacecraft and executes those with highest priority as soon as possible. The rest of the telecommands are forwarded to other components, depending on the destination of the relative service. This component also handles the critical events relating to both hardware and software. HK_FDIRManager manages the ICUSWs housekeeping service and the fault detection procedures. It notifies the EPD Manager of any detected critical event. It also executes telecommands related to the housekeeping service and the fault detection procedures. SensorTMManager captures the telemetry from the sensors and forwards it to the spacecraft onboard computer. The telemetry transmission rate is controlled according to a maximum limit. SensorTMManager executes, also, the telecommands related to the science service. Finnaly, BKG_TC_Executor executes the background telecommands forwarded by the EPDManager component.

EPDManager, HK_FDIR Manager and SensorTM Manager tasks require the use of periodic timers. The EDPManager task is subscribed to the run-time excep-

tion service through an exception port. This means that it receives a message whenever an exception occurs. The four tasks also access the shared `SCTxCannelCtrl` resource with the aims of sending the telemetry generated by the spacecraft. Figure 3 shows the time and exception ports required for each task, as well as the topology of the communication between tasks through the connection between the ports. The exchange of information between the tasks takes the form of either asynchronous or synchronous messages.

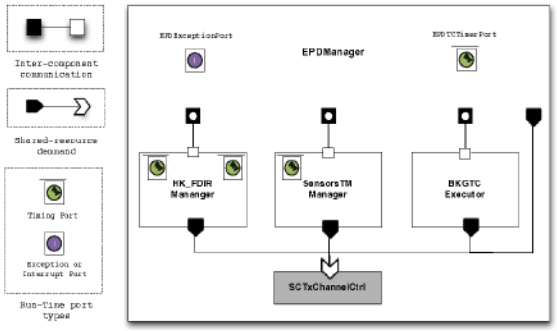

Figure 3. ICUSW component topology.

The set of the extra-functional properties related to the systems response time requires the appropriate assignment of task priorities. Specifically for the ICUSW the following priorities (P) have been established between tasks: $P(EPDManager) > P(HK\_FDIRManager) > P(SensorTM\_Manager) > P(BKGTC\_Executor)$

Figure 4 shows the `TSAMEvent` associated with the reception of science telecommands by EPD and the `TSAMMessageHandler` describing the reaction. This allows us to understand how the reactions are described in the transactional model. The periodic event sends a timeout message to the `EPDManager` component that triggers the capture of the reception buffers telecommands and their subsequent handling. The telecommands received by the missions onboard computer are stored in memory by a dedicated DMA-based hardware module.

Figure 5 shows the `TSAMEvent` associated with the periodic management of the telemetry by the SensorTM Manager component. This component manages the telemetry from the sensors, encapsulating the packets to the spacecraft computer. The telemetry can be generated in different ways depending on the real-time situation. Three real-time situation have been defined: 1) nominal, representing the normal state of operating state; 2) configuration, that corresponds to the instruments configuration; and 3) singular science event, corresponding to the occurrence of a science event of particular interest. This work only focuses on nominal real-time and singular science event situations.

The real-time situation associated with the singular science event is the one related to the telecommand (TC) 42

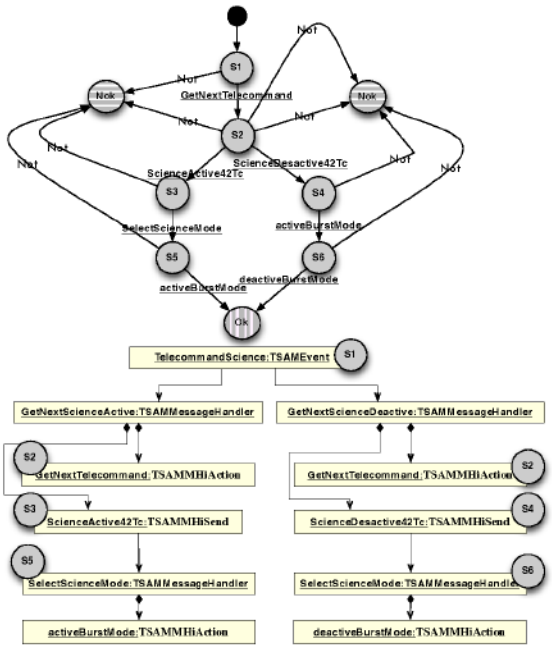

Figure 4. Top picture shows the Rapicheck FTA for End-to-End profile. Bottom picture shows the `TSAMEvent` and `TSAMMessageHandler` for retriving science telecommands.

(TC-42) function. The main function of TC-42 is to share scientific information among Solar Orbiters instruments. Each instrument can define a number of thresholds for activating a local singular science event mode based on information generated by other instruments. This process involves four steps: 1) periodically, each instrument sends four bits of information to the onboard computer, 2) the onboard computer generates TC-42. This TC contains all the information provided by all instruments, 3) TC-42 is broadcast to all the instruments, 4) the instruments act according to the information contained in TC-42.

Once TC-42 has been received and checked by the ICUSW, depending on the defined thresholds, it can activate a singular science mode. Within this mode, the generation of telemetry packets changes. The sensors telemetry generation rate is related to a Pulse Per Second (PPS) signal generated by the ICU and this is kept constant over time. If singular science event mode is active, data are time tagged with one second resolution; otherwise, a ten second resolution is used. The transition back to nominal mode is done after a predefined time period. Thus two RT-requirement models have been generated and two modulation points have been defined.

Figure 4 shows the two FTA related to the End-to-End-flow profile for the `TSAMEvent` managing the science telecommands. We can see how the `TSAMMHitems` which are shared in different `TSAMMessageHandler` are composed in the same state, as `getNextTC`. The `TSAMMHISend TCToSensorManager` is re-

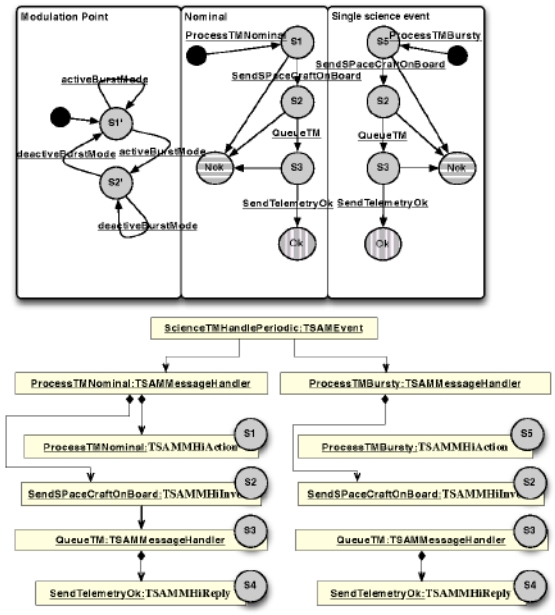

Figure 5. Top picture shows the Rapicheck FTA, the modulation point FTA verifies the crossed between the both real-time situation, the nominal and Single science event verify the right `TSAMMessageHandler` *for each real time situation. Bottom picture shows the* `TSAMEvent` *and* `TSAMMessageHandler` *for periodic management of science telemetry*

solved and placed following completion of all the `TSAMMHitems` of the `TSAMMessageHandler` `EDPManagerGetNextTc`. This is because they have a lower priority than `EPDManager`. With this profile we will be able to check that the sequence of actions for the execution of science telecommands is correct.

Figure 5 shows the two real-time situations implemented in the ICUSW being checked. The nominal state is the initial state; the transition from one state to another is performed by executing TC-42 for activation and deactivation. The image also shows the FTA concerning the `TSAMMessageHandler` states for the `SensorManager` and `TSAMEvent` `HandleSensorTelemetry` components. We can see that each one has different items related to the telemetry management.

7. CONCLUSIONS

In this work we have shown a process for automatic generation of timing constraints that can be verified automatically through the execution of a system on target. The timing constraints are generated from attributes in the design and the models, and the transformation rules of these models. This allows additional evidence to be produced proving that the transformations are correct and that the implementation behaves as expected. Of particular interest is the automatic verification of the schedulability analysis assumptions. Real life examples that are part of the flight software for the Instrument Control Unit of the Energetic Particle Detector on-board Solar Orbiter are used to ilustrate its applicability.

REFERENCES

[1] ECSS Secretariat, "Telemetry and telecommand packet utilization." ECSS-E-ST-40C, March 2009.

[2] S. Kent, "Model driven engineering," in *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, (London, UK, UK), pp. 286–298, Springer-Verlag, 2002.

[3] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2nd ed., 2002.

[4] B. W. Boehm, "Verifying and validating software requirements and design specifications," *IEEE Software*, pp. 75–88, 1984.

[5] M. G. Harbour, J. G. García, J. P. Gutiérrez, and J. D. Moyano, "Mast: Modeling and analysis suite for real time applications," *Real-Time Systems, Euromicro Conference on*, vol. 0, p. 0125, 2001.

[6] O. R. Polo, S. Esteban, A. Grau, and J. M. de la Cruz, "Control code generator used for control experiments in ship scale model," in *Proceedings of the CAMS2001 IFAC Conference*, 2001.

[7] T. Bures, P. Hnetynka, and F. Plasil, "Sofa 2.0: Balancing advanced features in a hierarchical component model," in *Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, (Washington, DC, USA), pp. 40–48, IEEE Computer Society, 2006.

[8] B. Selic, G. Gullekson, and P. T. Ward, *Real-time object-oriented modeling*. New York, NY, USA: John Wiley & Sons, Inc., 1994.

[9] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, pp. 231–274, June 1987.

[10] B. Selic, "Uml 2: a model-driven development tool," *IBM Systems Journal*, vol. 45, no. 3, pp. 607–620, 2006.

[11] P. Parra and O. R. Polo, "MICOBS: multi-platform multi-model component based software development framework," in *Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering*, CBSE '11, (New York, NY, USA), pp. 1–10, ACM, 2011.